5ª edición

Sistemas operativos

Aspectos internos y principios de diseño





Sistemas operativos

Aspectos internos y principios de diseño

Quinta Edición

WILLIAM STALLINGS

Traducción y revisión técnica

José María Peña Sánchez
Fernando Pérez Costoya
María de los Santos Pérez Hernández
Víctor Robles Forcada
Francisco Javier Rosales García

Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid



Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima
Montevideo • San Juan • San José • Santiago • Sâo Paulo • White Plains

Datos de catalogación bibliográfica

SISTEMAS OPERATIVOS
William Stallings

Pearson Educación, S.A., Madrid, 2005

ISBN: 978-84-205-5796-0

Materia: Informática 681.3

Formato: 195 x 250 mm. **Páginas: 872**

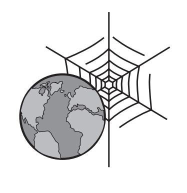
SISTEMAS OPERATIVOS
William Stallings
ISBN: 84-205-4462-0
Depósito Legal:
PEARSON PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN S.A.

Equipo editorial
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

Equipo de producción
Director: José A. Clares
Técnico: Isabel Muñoz

Diseño de cubierta
Equipo de diseño de Pearson Educación, S.A.

Impreso por
IMPRESO EN ESPAÑA - PRINTED IN SPAIN
Este libro ha sido impreso con papel y tintas ecológicos



PÁGINA WEB PARA *SISTEMAS OPERATIVOS: ASPECTOS INTERNOS Y PRINCIPIOS DE DISEÑO, QUINTA EDICIÓN*

La página web en WilliamStallings.com/OS/OS5e.html proporciona apoyo a profesores y estudiantes que utilicen este libro. Incluye los siguientes elementos.

MATERIAL PARA APOYO DE CURSOS

El material para apoyo de los cursos incluye:

- Copia de las figuras del libro en formato PDF.
- Copia de las tablas del libro en formato PDF.
- Un conjunto de transparencias PowerPoint para utilizarlas como ayuda en clase.
- Notas de clase en HTML que pueden servir como material de ayuda para el estudio.
- **Página de Recursos del Estudiante de Informática (*Computer Science Student Resource Site*)**: contienen gran número de enlaces y documentos que los estudiantes pueden encontrar útiles para su formación en informática. Esta página incluye una revisión de las matemáticas básicas relacionadas; consejos para la búsqueda de información, redacción, y realización de problemas en casa; enlaces a repositorios de información de informática, tales como informes y bibliografías; y otros enlaces de interés.
- Una hoja de erratas del libro, actualizada casi mensualmente.

DOCUMENTOS COMPLEMENTARIOS

Los documentos complementarios incluyen:

- Una copia en PDF de todos los algoritmos del libro en un pseudo-código de tipo Pascal de fácil lectura.
- Material del libro relativo a Windows, UNIX, y Linux; reproducido en tres documentos PDF de fácil referencia.
- Varios documentos que amplían lo tratado en el libro. Incluye aspectos relativos a la complejidad de los algoritmos, estándares de Internet y *Sockets*.

CURSOS DE SISTEMAS OPERATIVOS

La página web de OS5e incluye enlaces a otras páginas de cursos impartidos usando este libro. Estas páginas pueden proporcionar guías útiles sobre cómo planificar y ordenar los temas tratados, así como un gran número de anotaciones y material diverso.

PÁGINAS WEB ÚTILES

La página web de OS5e incluye también enlaces a otras páginas de interés. Los enlaces cubren un amplio espectro de temas y permitirán a los estudiantes explorar aspectos concretos con gran profundidad.

LISTA DE CORREO ELECTRÓNICO

Se mantiene una lista de correo para que los profesores que utilicen este libro puedan intercambiar información, sugerencias, y preguntas entre ellos y con el autor. La información de suscripción se proporciona en la página web del libro.

PROYECTOS DE SISTEMAS OPERATIVOS

La página web incluye enlaces a las páginas de Nachos y BACI, que son dos paquetes software que sirven como entornos para implementación de proyectos. Cada página incluye software para descargar con información de apoyo. *Véase* el Apéndice C para más información.

Contenido

SEGUNDA PARTE: PROCESOS 105

Prólogo

OBJETIVOS

Este libro se ocupa de los conceptos, la estructura y los mecanismos de los sistemas operativos. Su propósito es presentar, de la manera más clara y completa posible, la naturaleza y las características de los sistemas operativos de hoy en día.

Esta tarea es un reto por varios motivos. En primer lugar, los computadores para los que se diseñan los sistemas operativos presentan una enorme variedad. Esta diversidad incluye desde estaciones de trabajo y computadores personales para un único usuario, pasando por sistemas compartidos de tamaño medio, hasta grandes sistemas *mainframe* y supercomputadores, así como máquinas especializadas tales como los sistemas de tiempo real. La variedad no está sólo en la capacidad y la velocidad de las máquinas, sino también en los requisitos de las aplicaciones y del sistema. En segundo lugar, el rápido ritmo de cambios que ha caracterizado siempre a los sistemas informáticos continúa sin remitir. Diversas áreas fundamentales en el diseño de sistemas operativos son de reciente aparición, estando todavía activa la investigación sobre las mismas, así como sobre otras nuevas áreas.

A pesar de esta variedad y de este ritmo de cambios incesante, ciertos conceptos fundamentales siguen siendo aplicables en todo momento. Evidentemente, su aplicación depende del estado actual de la tecnología y de los requisitos particulares de la aplicación. El objetivo de este libro es proporcionar un estudio profundo de los fundamentos del diseño de sistemas operativos y relacionarlos con aspectos de diseño contemporáneos y con las tendencias actuales en el desarrollo de sistemas operativos.

SISTEMAS DE EJEMPLO

Este libro está destinado a dar a conocer al lector los principios de diseño y los aspectos de implementación de los sistemas operativos contemporáneos. Por consiguiente, un tratamiento puramente conceptual o teórico sería inadecuado. Para mostrar los conceptos y asociarlos a alternativas de diseño del mundo real, se han seleccionado tres sistemas operativos como ejemplos reales:

- **Windows XP y Windows 2003.** Un sistema operativo multitarea para computadores personales, estaciones de trabajo y servidores. Al tratarse de un nuevo sistema operativo, incorpora de una manera nítida muchos de los últimos desarrollos en la tecnología de sistemas operativos. Además, Windows es uno de los primeros sistemas operativos comerciales importantes que está estrechamente basado en principios de diseño orientado a objetos. Este libro se ocupa de la tecnología utilizada en las versiones más recientes de Windows, XP para estaciones de trabajo y computadores personales, y 2003 para servidores.
- **UNIX.** Un sistema operativo multiusuario, originalmente destinado a minicomputadores, pero implementado en un amplio rango de máquinas desde poderosos microcomputadores a supercomputadores. Se incluyen dos versiones de UNIX. UNIX SVR4 es un sistema muy usado que incorpora muchas características avanzadas. Solaris es la versión comercial más utilizada de UNIX. Incluye procesamiento multihilo y otras características que no se encuentran en SVR4 ni en la mayoría de las otras versiones de UNIX.
- **Linux.** Una versión de UNIX cuyo código fuente está disponible libremente, que es muy utilizada actualmente.

Estos sistemas se seleccionaron por su relevancia y representatividad. El estudio de los sistemas de ejemplo se distribuye a lo largo del texto en vez de agruparlos en un solo capítulo o apéndice. Así, durante el estudio de la concurrencia, se describen los mecanismos de concurrencia de cada sistema de ejemplo, y se explica la motivación de las diversas opciones de diseño individuales. Con este enfoque, los conceptos de diseño estudiados en cualquier capítulo se refuerzan inmediatamente con ejemplos del mundo real.

AUDIENCIA A LA QUE ESTÁ DESTINADO

Este libro está destinado tanto a una audiencia de carácter académico como a una de perfil profesional. Como libro de texto, está pensado para un curso de sistemas operativos de un semestre para las titulaciones de Informática, Ingeniería de Computadores e Ingeniería Eléctrica. Incluye los temas recomendados por el *Computer Curricula 2001* para programas universitarios de informática, realizado por el equipo de trabajo conjunto para planes de estudio de informática (*Joint Task Force on Computing Curricula*) de la sociedad informática (*Computer Society*) de IEEE y ACM. El libro también trata los temas recomendados por *Guidelines for Associate-Degree Curricula in Computer Science 2002*, también del equipo de trabajo conjunto para planes de estudio de informática de la sociedad informática de IEEE y ACM. El libro sirve igualmente como un volumen de referencia básico, adecuado para el estudio personal.

ORGANIZACIÓN DEL LIBRO

Este libro se divide en seis partes (*véase* el Capítulo 0 para una visión general):

- Antecedentes.
- Procesos.
- Memoria.
- Planificación.
- Entrada/salida y ficheros.
- Sistemas distribuidos y seguridad.

Este libro incluye diversas características pedagógicas, como el uso de numerosas figuras y tablas para facilitar el estudio. Cada capítulo incluye una lista de términos clave, preguntas de repaso, problemas, propuestas de lecturas adicionales y direcciones de sitios web relevantes. Además, está disponible para los profesores una batería de preguntas de test.

SERVICIOS DE INTERNET PARA PROFESORES Y ESTUDIANTES

Hay un sitio web asociado a este libro que proporciona apoyo a los estudiantes y a los profesores. El sitio incluye enlaces a otros sitios relevantes, copias originales de las transparencias de las figuras y tablas del libro en formato PDF (Adobe Acrobat), transparencias en PowerPoint e información para darse de alta en la lista de correo de Internet del libro. La página web está en WilliamStallings.com/OS/OS5e.html. *Véase* la Sección "Sitio web de *sistemas operativos. Aspectos internos y principios de diseño*" anterior a este prólogo para más información. Se ha establecido una lista de correo para que los profesores que usan este libro puedan intercambiar información, sugerencias y preguntas entre sí y con el propio autor. En cuanto se descubran errores tipográficos o de otro tipo, se

publicará una lista de erratas en WilliamStallings.com. Por último, hay que resaltar que el autor mantiene un sitio para el estudiante de informática en WilliamStallings.com/StudentSupport.html.

PROYECTOS DE SISTEMAS OPERATIVOS

Para muchos instructores, un elemento importante de un curso de sistemas operativos es un proyecto o un conjunto de proyectos mediante los cuales el estudiante obtiene una experiencia práctica que le permite reforzar los conceptos del libro. Este libro proporciona un incomparable grado de apoyo en ese aspecto, incluyendo un componente de proyectos en el curso. En el interior del libro se definen dos proyectos de programación principales. El sitio web del profesor ofrece referencias en línea que pueden utilizar los estudiantes para abordar estos proyectos de forma gradual. Se proporciona información sobre tres paquetes de software que sirven como entornos de trabajo para la implementación de proyectos: OSP y NACHOS para desarrollar componentes de un sistema operativo, y BACI para estudiar los mecanismos de concurrencia. Además, el sitio web del profesor incluye una serie de pequeños proyectos de programación, cada uno pensado para desarrollarse en una o dos semanas, que cubre un amplio rango de temas y que pueden implementarse en cualquier lenguaje apropiado y sobre cualquier plataforma, así como proyectos de investigación y tareas de lectura y análisis. *Véase* los apéndices para más detalles.

NOVEDADES DE LA QUINTA EDICIÓN

En esta nueva edición, el autor ha intentado recoger las innovaciones y mejoras que ha habido en esta disciplina durante los cuatro años que han transcurrido desde la última edición, manteniendo un tratamiento amplio y completo de esta materia. Asimismo, varios profesores que imparten esta disciplina, así como profesionales que trabajan en este campo, han revisado en profundidad la cuarta edición. Como consecuencia de este proceso, en muchas partes del libro, se ha mejorado la claridad de la redacción y de las ilustraciones que acompañan al texto. Además, se han incluido varios problemas de carácter realista.

Además de mejoras pedagógicas y en su presentación de cara al usuario, el contenido técnico del libro se ha actualizado completamente, para reflejar los cambios actuales en esta excitante disciplina. El estudio de Linux se ha extendido significativamente, basándose en su última versión: Linux 2.6. El estudio de Windows se ha actualizado para incluir Windows XP y Windows Server 2003. Se ha revisado y extendido el material dedicado a la concurrencia para mejorar su claridad, moviendo parte del mismo a un apéndice, e incluyendo un estudio sobre condiciones de carrera. El tratamiento de la planificación en esta nueva versión incluye un estudio de la inversión de prioridades. Hay un nuevo capítulo sobre redes, presentándose el API de Sockets. Además, se ha ampliado el tratamiento del diseño orientado a objetos.

AGRADECIMIENTOS

Esta nueva edición se ha beneficiado de la revisión realizada por diversas personas, que aportaron generosamente su tiempo y experiencia. Entre ellos se incluyen Stephen Murrell (Universidad de Miami), David Krumme (Universidad de Tufts), Duncan Buell (Universidad de Carolina), Amit Jain (Universidad de Bosie State), Fred Kuhns (Universidad de Washington, St.Louis), Mark McCullen (Universidad de Michigan State), Jayson Rock (Universidad de Wisconsin-Madison), David Middleton (Universidad de Arkansas Technological) y Binhai Zhu (Universidad de Montana State), todos revisaron la mayor parte o todo el libro.

El autor da las gracias también a mucha gente que realizó revisiones detalladas de uno o más capítulos: Javier Eraso Helguera, Andrew Cheese, Robert Kaiser, Bhavin Ghandi, Joshua Cope, Luca Ve-

nuti, Gregory Sharp, Marisa Gil, Balbir Singh, Mrugesh Gajjar, Bruce Janson, Mayan Moudgill, Pete Bixby, Sonja Tideman, Siddharth Choudhuri, Zhihui Zhang, Andrew Huo Zhigang, Yibing Wang, Darío Álvarez y Michael Tsai. Asimismo, al autor le gustaría agradecer a Tigran Aivazian, autor del documento sobre los aspectos internos del núcleo de Linux (*Linux Kernel Internals*), que es parte del proyecto de documentación de Linux (*Linux Documentation Project*), por su revisión del material sobre Linux 2.6. Ching-Kuang Shene (Universidad de Michigan Tech) proporcionó los ejemplos usados en la sección sobre condiciones de carrera y revisó dicha sección.

Asimismo, Fernando Ariel Gont contribuyó con diversos ejercicios para el estudiante y llevó a cabo revisiones detalladas de todos los capítulos.

El autor querría también dar las gracias a Michael Kifer y Scott A. Smolka (SUNY–Stony Brook) por contribuir al Apéndice D, a Bill Bynum (College of William and Mary) y Tracy Camp (Colorado School of Mines) por prestar su ayuda en el Apéndice E; Steve Taylor (Worcester Polytechnic Institute) por colaborar en los proyectos de programación y en las tareas de lectura y análisis del manual del profesor, y al profesor Tan N. Nguyen (Universidad de George Mason) por contribuir a los proyectos de investigación del manual del profesor. Ian G.Graham (Universidad de Griffith) colaboró con los dos proyectos de programación del libro. Oskars Rieksts (Universidad de Kutztown) permitió de forma generosa que se hiciera uso de sus notas de clase, ejercicios y proyectos.

Por último, el autor querría dar las gracias a las numerosas personas responsables de la publicación del libro, todas realizaron como de costumbre un excelente trabajo. Esto incluye al personal de Prentice Hall, particularmente a los editores Alan Apt y Toni Holm, su ayudante Patrick Lindner, la directora de producción Rose Kernan, y la directora de suplementos Sarah Parker. Este agradecimiento se extiende también a Jake Warde de Warde Publishers que dirigió el proceso de revisión, y a Patricia M. Daly que realizó la edición de la copia.

CAPÍTULO 0

Guía del lector

◆ ◆

Este libro, junto con su sitio web asociado, cubre una gran cantidad de material. A continuación, se le proporciona al lector una visión general del mismo.

◆ ◆

0.1. ORGANIZACIÓN DEL LIBRO

El libro está organizado en siete partes:

Primera parte. Antecedentes. Proporciona una introducción a la arquitectura y organización del computador, haciendo énfasis en aquellos aspectos relacionados con el diseño de sistemas operativos, presentando, asimismo, una visión general de los temas de sistemas operativos tratados en el resto del libro.

Segunda parte. Procesos. Presenta un análisis detallado de los procesos, el procesamiento multihilo, el multiprocesamiento simétrico (*Symmetric Multiprocessing*, SMP) y los micronúcleos. En esta parte se estudian también los aspectos principales de la concurrencia en un sistema uniprocesador, haciendo hincapié en los temas de la exclusión mutua y de los interbloqueos.

Tercera parte. Memoria. Proporciona un extenso estudio de las técnicas de gestión de memoria, incluyendo la memoria virtual.

Cuarta parte. Planificación. Ofrece un estudio comparativo de diversas estrategias de planificación de procesos. Se examinará también la planificación de hilos, de SMP y de tiempo real.

Quinta parte. Entrada/salida y ficheros. Examina los aspectos involucrados en el control de las operaciones de E/S por parte del sistema operativo. Se dedica especial atención a la E/S del disco, que es fundamental para el rendimiento del sistema. Asimismo, proporciona una visión general de la gestión de ficheros.

Sexta parte. Sistemas distribuidos y seguridad. Estudia las principales tendencias en redes de computadores, incluyendo TCP/IP, procesamiento cliente/servidor y *clusters*. Asimismo, describe algunas áreas de diseño fundamentales en el desarrollo de los sistemas operativos distribuidos. El Capítulo 16 proporciona un estudio de las amenazas y los mecanismos para proporcionar seguridad al computador y a la red.

Este libro está dedicado a dar a conocer a los lectores los principios de diseño y los aspectos de implementación de los sistemas operativos contemporáneos. Por tanto, sería inadecuado un tratamiento puramente teórico o conceptual. Para mostrar los conceptos y asociarlos a opciones de diseño que se deben tomar en la vida real, se han seleccionado dos sistemas operativos como ejemplos reales:

- **Windows.** Un sistema operativo multitarea diseñado para ejecutar en diversos computadores personales, estaciones de trabajo y servidores. Es uno de los pocos sistemas operativos comerciales recientes diseñado esencialmente desde cero. Debido a esto, está en una buena posición para incorporar de una manera nítida los más recientes desarrollos en la tecnología de sistemas operativos.
- **UNIX.** Un sistema operativo multitarea destinado originalmente a minicomputadores pero implementado en un amplio rango de máquinas desde poderosos microprocesadores a supercomputadores. Dentro de esta familia de sistemas operativos, se incluye Linux.

El estudio de los sistemas de ejemplo está distribuido a través del libro en vez de agrupado en un único capítulo o apéndice. Así, durante el estudio de la concurrencia, se describe el mecanismo de concurrencia de cada sistema de ejemplo y se discute la motivación de las opciones de diseño particulares. Con esta estrategia, los conceptos de diseño estudiados en un determinado capítulo son inmediatamente reforzados con los ejemplos del mundo real.

0.2. ORDEN DE PRESENTACIÓN DE LOS TEMAS

Sería natural que los lectores cuestionaran el orden particular de presentación de los temas en este libro. Por ejemplo, el tema de planificación (Capítulos 9 y 10) está muy relacionado con los dedicados a la concurrencia (Capítulos 5 y 6) y el tema general de procesos (Capítulo 3), por lo que podría ser razonable tratarlo inmediatamente después de estos temas.

La dificultad reside en que los diversos temas están estrechamente interrelacionados. Por ejemplo, para tratar la memoria virtual, es útil hacer referencia a los aspectos de planificación relacionados con un fallo de página. Por otro lado, también es útil referirse a algunos aspectos de gestión de memoria cuando se estudian decisiones de planificación. Este tipo de ejemplo se puede repetir indefinidamente: El estudio de la planificación requiere algunos conocimientos de la gestión de E/S y viceversa.

La Figura 0.1 sugiere algunas relaciones importantes entre los temas. Las líneas continuas indican relaciones muy estrechas, desde el punto de vista de las decisiones de diseño y de implementación. Basados en este diagrama, es razonable comenzar con una discusión básica de procesos, que corresponde con el Capítulo 3. Después de eso, el orden puede ser un poco arbitrario. Muchos tratados de sistemas operativos reúnen todo el material sobre procesos al principio y después tratan otros te-

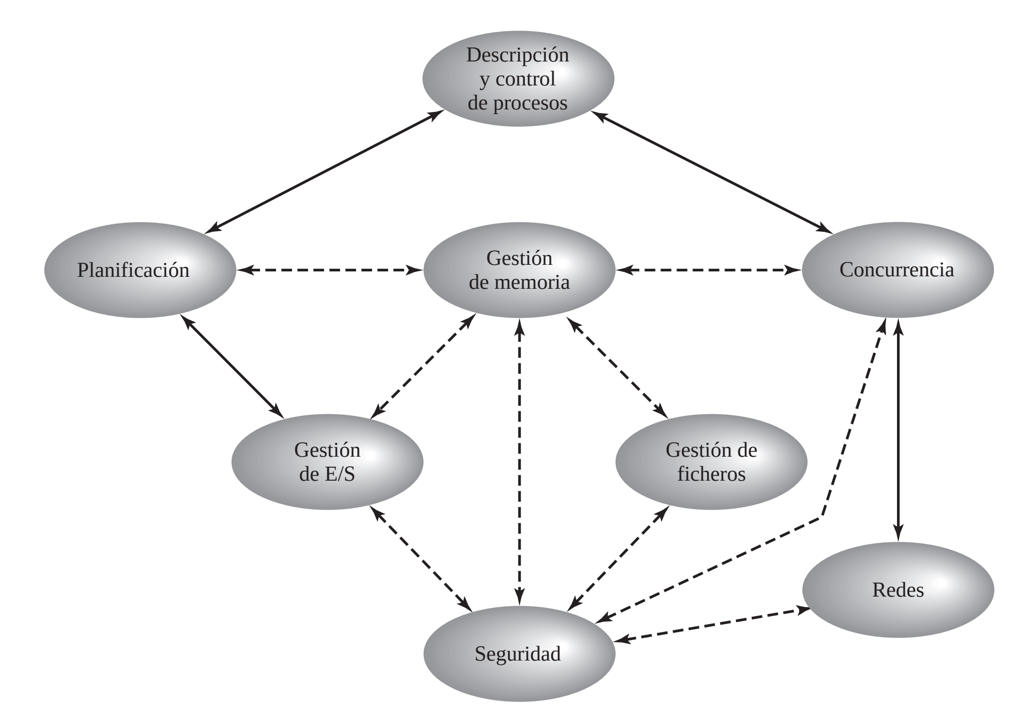


Figura 0.1. Temas de sistemas operativos.

mas. Esto es ciertamente válido. Sin embargo, la importancia fundamental de la gestión de memoria, que en opinión del autor es tan importante como la gestión de procesos, ha llevado a la decisión de presentar este tema antes de profundizar en la planificación.

La solución ideal es que el estudiante, después de completar los Capítulos del 1 al 3 en ese orden, lea y asimile los capítulos siguientes en paralelo: el 4 seguido (opcionalmente) del 5; el 6 seguido por el 7; el 8 seguido (opcionalmente) del 9; y el 10. Por último, se pueden estudiar los siguientes capítulos en cualquier orden: el 11; el 12 seguido del 13; el 14; y el 15. Sin embargo, aunque el cerebro humano puede llevar a cabo un procesamiento paralelo, al estudiante le resulta imposible (y caro) trabajar con éxito simultáneamente con cuatro copias del mismo libro abiertas en cuatro capítulos diferentes. Dada la necesidad de un orden lineal, el autor considera que el orden utilizado en este libro es el más efectivo.

0.3. RECURSOS EN INTERNET Y EN LA WEB

Hay diversos recursos disponibles en Internet y en la Web para apoyar a este libro y ayudar al lector a mantenerse al día con los avances en este campo.

SITIOS WEB DE ESTE LIBRO

Se ha creado una página web especial para este libro en **WilliamStallings.com/OS/OS5e.html**. Consulte el diagrama de dos páginas al principio de este libro para obtener una descripción detallada de este sitio web. De especial interés son los dos documentos disponibles en el sitio web para el estudiante:

- **Pseudo-código.** Para los lectores no acostumbrados al lenguaje C, se reproducen todos los algoritmos también en un pseudo-código similar al Pascal. Este lenguaje de pseudo-código es intuitivo y particularmente fácil de seguir.
- **Descripciones de Windows, UNIX y Linux.** Como se ha mencionado previamente, se utilizan Windows y diversas versiones de UNIX como ejemplos de casos reales, estando este estudio distribuido a través del texto en vez de agrupado en un único capítulo o apéndice. Algunos lectores preferirían tener todo este material en un único sitio para usarlo como referencia. Por tanto, todo el material de Windows y UNIX del libro se reproduce en tres documentos en el sitio web.

En cuanto se detecte cualquier error tipográfico o de otro tipo, se publicará una lista de erratas de este libro en el sitio web. Por favor, informe de cualquier error que detecte en el libro. En **William Stallings.com** se encuentran las hojas de erratas de los otros libros publicados por el autor, así como información sobre descuentos en pedidos de libros.

También se mantiene un sitio con recursos para el estudiante de informática (*Computer Science Student Resource Site*), en **WilliamStallings.com/StudentSupport.html**; el objetivo de este sitio es proporcionar documentos, información y enlaces para estudiantes de informática. Los enlaces se organizan en cuatro categorías:

- **Matemáticas.** Incluye un repaso sobre matemáticas básicas, una introducción al análisis de colas y a los sistemas númericos, así como enlaces a numerosos sitios con información sobre matemáticas.
- ***How-to*.** Aconseja y guía al estudiante para resolver sus ejercicios, escribir informes técnicos y preparar presentaciones técnicas.

- **Recursos de investigación.** Proporciona enlaces a recopilaciones importantes de artículos, informes técnicos y referencias bibliográficas.
- **Misceláneos.** Incluye diversos documentos y enlaces útiles.

OTROS SITIOS WEB

Hay numerosos sitios web que proporcionan información relacionada con los temas tratados en este libro. En los siguientes capítulos, pueden encontrarse referencias a sitios web específicos en la Sección «Lecturas recomendadas». Debido a que el URL de un sitio web particular puede cambiar, este libro no incluye direcciones URL. En el sitio web de este libro puede encontrarse el enlace apropiado de todos los sitios web nombrados en el libro.

GRUPOS DE NOTICIAS DE USENET

Diversos grupos de noticias de USENET se dedican a algún tema relacionado con los sistemas operativos o con un determinado sistema operativo. Como ocurre prácticamente con todos los grupos de USENET, hay un alto porcentaje de ruido en la señal, pero es un experimento valioso comprobar si alguno satisface las necesidades del lector. Los más relevantes son los siguientes:

- **comp.os.research.** El grupo que más interesa seguir. Se trata de un grupo de noticias moderado que se dedica a temas de investigación.
- **comp.os.misc.** Un foro de discusión general sobre temas de sistemas operativos.
- **comp.unix.internals**
- **comp.os.linux.development.system**

PARTE I

ANTECEDENTES

En esta primera parte se proporcionan los antecedentes necesarios y se establece el contexto para el resto de este libro, presentando los conceptos fundamentales sobre arquitectura de computadores y sobre los aspectos internos de los sistemas operativos.

GUÍA DE LA PRIMERA PARTE

CAPÍTULO 1. INTRODUCCIÓN A LOS COMPUTADORES

Un sistema operativo hace de intermediario entre, por un lado, los programas de aplicación, las herramientas y los usuarios, y, por otro, el hardware del computador. Para apreciar cómo funciona el sistema operativo y los aspectos de diseño involucrados, se debe tener algún conocimiento de la organización y la arquitectura de los computadores. El Capítulo 1 proporciona un breve estudio del procesador, la memoria y los elementos de E/S de un computador.

CAPÍTULO 2. INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

El tema del diseño de un sistema operativo (S.O.) abarca un enorme campo, resultando fácil enredarse en los detalles, perdiendo el contexto general durante el estudio de un tema en particular. El Capítulo 2 proporciona una visión general a la que el lector puede volver en cualquier punto del libro para recuperar el contexto global. Se comienza con una exposición de los objetivos y funciones del sistema operativo. A continuación, por su relevancia histórica, se describen algunos sistemas y funciones del S.O. Este estudio permite presentar algunos principios de diseño del S.O. fundamentales en un entorno sencillo, de manera que queden claras las relaciones entre varias funciones del S.O. A continuación, el capítulo resalta las características más importantes de los sistemas operativos modernos. A lo largo de este libro, cuando se presentan diversos temas, es necesario hablar tanto de principios fundamentales y bien consolidados como de las más recientes innovaciones en el diseño de SS.OO. El análisis de este capítulo hace notar al lector que se debe de abordar esta mezcla de técnicas de diseño ya consolidadas con otras recientes. Finalmente, se presenta una introducción de Windows y UNIX; este estudio establece la arquitectura general de estos sistemas, proporcionando un contexto para las discusiones detalladas que se realizan más adelante.

CAPÍTULO 1

Introducción a los computadores

◆ ◆

Un sistema operativo explota los recursos hardware de uno o más procesadores para proporcionar un conjunto de servicios a los usuarios del sistema. El sistema operativo también gestiona la memoria secundaria y los dispositivos de E/S (entrada/salida) para sus usuarios. Por tanto, es importante tener algunos conocimientos del hardware del computador subyacente antes de iniciar el estudio de los sistemas operativos.

Este capítulo proporciona una visión general del hardware del computador. En la mayoría de las áreas, el estudio es breve, asumiendo que el lector ya está familiarizado con este tema. Sin embargo, se estudiarán con cierto detalle varios aspectos por su repercusión en los temas tratados más adelante en el libro.

◆ ◆

1.1. ELEMENTOS BÁSICOS

Al más alto nivel, un computador consta del procesador, la memoria y los componentes de E/S, incluyendo uno o más módulos de cada tipo. Estos componentes se interconectan de manera que se pueda lograr la función principal del computador, que es ejecutar programas. Por tanto, hay cuatro elementos estructurales principales:

- **Procesador.** Controla el funcionamiento del computador y realiza sus funciones de procesamiento de datos. Cuando sólo hay un procesador, se denomina usualmente **unidad central de proceso** (*Central Processing Unit*, CPU).

- **Memoria principal.** Almacena datos y programas. Esta memoria es habitualmente volátil; es decir, cuando se apaga el computador, se pierde su contenido. En contraste, el contenido de la memoria del disco se mantiene incluso cuando se apaga el computador. A la memoria principal se le denomina también *memoria real* o *memoria primaria*.

- **Módulos de E/S.** Transfieren los datos entre el computador y su entorno externo. El entorno externo está formado por diversos dispositivos, incluyendo dispositivos de memoria secundaria (por ejemplo, discos), equipos de comunicaciones y terminales.

- **Bus del sistema.** Proporciona comunicación entre los procesadores, la memoria principal y los módulos de E/S.

La Figura 1.1 muestra estos componentes de más alto nivel. Una de las funciones del procesador es el intercambio de datos con la memoria. Para este fin, se utilizan normalmente dos registros internos (al procesador): un registro de dirección de memoria (RDIM), que especifica la dirección de memoria de la siguiente lectura o escritura; y un registro de datos de memoria (RDAM), que contiene los datos que se van a escribir en la memoria o que recibe los datos leídos de la memoria. De manera similar, un registro de dirección de E/S (RDIE/S) especifica un determinado dispositivo de E/S, y un registro de datos de E/S (RDAE/S) permite el intercambio de datos entre un módulo de E/S y el procesador.

Un módulo de memoria consta de un conjunto de posiciones definidas mediante direcciones numeradas secuencialmente. Cada posición contiene un patrón de bits que se puede interpretar como una instrucción o como datos. Un módulo de E/S transfiere datos desde los dispositivos externos hacia el procesador y la memoria, y viceversa. Contiene *buffers* (es decir, zonas de almacenamiento internas) que mantienen temporalmente los datos hasta que se puedan enviar.

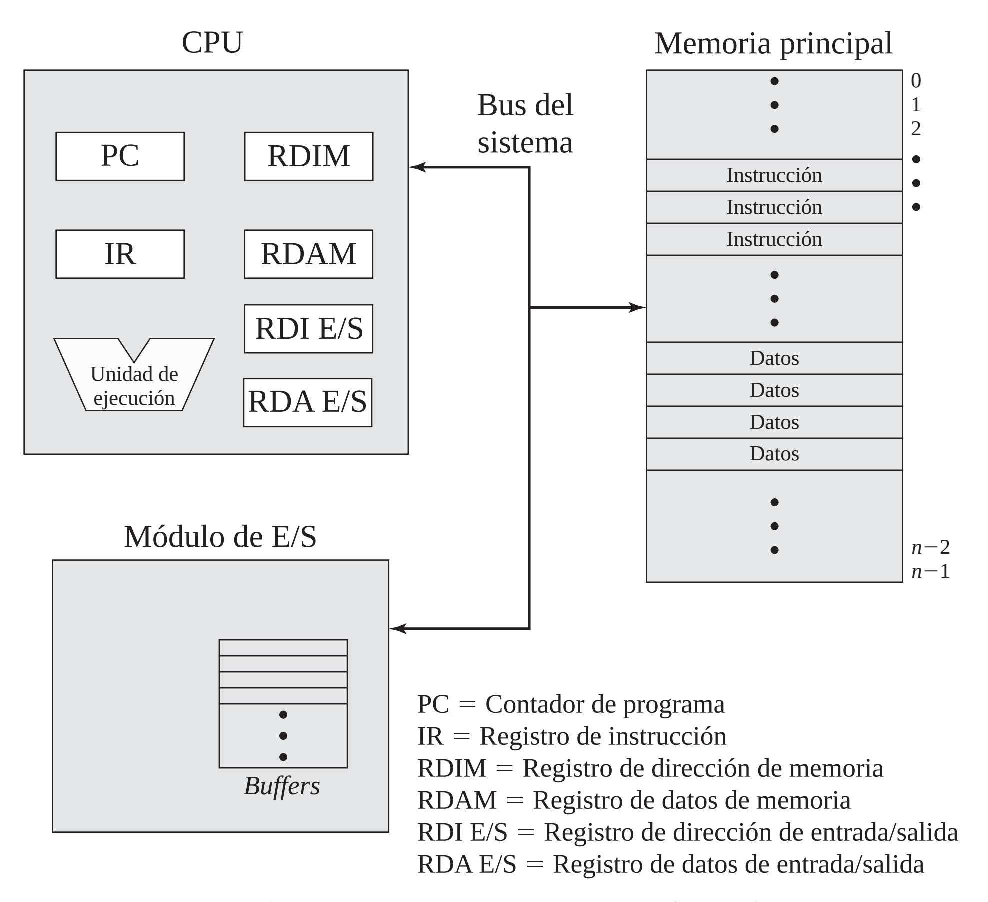


Figura 1.1. Componentes de un computador: visión al más alto nivel.

1.2. REGISTROS DEL PROCESADOR

Un procesador incluye un conjunto de registros que proporcionan un tipo de memoria que es más rápida y de menor capacidad que la memoria principal. Los registros del procesador sirven para dos funciones:

- **Registros visibles para el usuario.** Permiten al programador en lenguaje máquina o en ensamblador minimizar las referencias a memoria principal optimizando el uso de registros. Para lenguajes de alto nivel, un compilador que realice optimización intentará tomar decisiones inteligentes sobre qué variables se asignan a registros y cuáles a posiciones de memoria principal. Algunos lenguajes de alto nivel, tales como C, permiten al programador sugerir al compilador qué variables deberían almacenarse en registros.

- **Registros de control y estado.** Usados por el procesador para controlar su operación y por rutinas privilegiadas del sistema operativo para controlar la ejecución de programas.

No hay una clasificación nítida de los registros entre estas dos categorías. Por ejemplo, en algunas máquinas el contador de programa es visible para el usuario, pero en muchas otras no lo es. Sin embargo, para el estudio que se presenta a continuación, es conveniente utilizar estas categorías.

REGISTROS VISIBLES PARA EL USUARIO

A un registro visible para el usuario se puede acceder por medio del lenguaje de máquina ejecutado por el procesador que está generalmente disponible para todos los programas, incluyendo tanto programas de aplicación como programas de sistema. Los tipos de registros que están normalmente disponibles son registros de datos, de dirección y de códigos de condición.

El programador puede utilizar los **registros de datos** para diversas funciones. En algunos casos, son, en esencia, de propósito general y pueden usarse con cualquier instrucción de máquina que realice operaciones sobre datos. Sin embargo, frecuentemente, hay restricciones. Por ejemplo, puede haber registros dedicados a operaciones de coma flotante y otros a operaciones con enteros.

Los **registros de dirección** contienen direcciones de memoria principal de datos e instrucciones, o una parte de la dirección que se utiliza en el cálculo de la dirección efectiva o completa. Estos registros pueden ser en sí mismos de propósito general, o pueden estar dedicados a una forma, o modo, particular de direccionamiento de memoria. A continuación, se incluyen algunos ejemplos:

- **Registro índice.** El direccionamiento indexado es un modo común de direccionamiento que implica sumar un índice a un valor de base para obtener una dirección efectiva.
- **Puntero de segmento.** Con direccionamiento segmentado, la memoria se divide en segmentos, que son bloques de palabras[1] de longitud variable. Una referencia de memoria consta de una referencia a un determinado segmento y un desplazamiento dentro del segmento; este modo de direccionamiento es importante en el estudio de la gestión de memoria que se realizará en el Capítulo 7. En este modo de direccionamiento, se utiliza un registro para mantener la dirección base (posición de inicio) del segmento. Puede haber múltiples registros; por ejemplo, uno para el sistema operativo (es decir, cuando el código del sistema operativo se está ejecutando en el procesador) y otro para la aplicación que se está ejecutando actualmente.
- **Puntero de pila.** Si hay direccionamiento de pila[2] visible para el usuario, hay un registro dedicado que apunta a la cima de la pila. Esto permite el uso de instrucciones que no contienen campo de dirección, tales como las que permiten apilar (*push*) y extraer (*pop*).

En algunas máquinas, una llamada a una subrutina o a un procedimiento implica salvar automáticamente todos los registros visibles para el usuario, que se restaurarán al retornar. El procesador realiza estas operaciones de salvar y restaurar como parte de la ejecución de las instrucciones de llamada y de retorno. Esto permite que cada procedimiento use estos registros independientemente. En otras máquinas, el programador es el responsable de guardar el contenido de los registros visibles para el usuario antes de una llamada a un procedimiento, incluyendo instrucciones para ello en el programa. Por tanto, las funciones de salvar y restaurar se pueden realizar en hardware o en software, dependiendo del procesador.

---

[1] No hay una definición universal del término *palabra*. En general, una **palabra** es un conjunto ordenado de bytes o bits que es la unidad normal con la que se almacena, transmite, u opera la información dentro de un determinado computador. Normalmente, si un computador tiene un juego de instrucciones de longitud fija, la longitud de las instrucciones es igual a la de la palabra.

[2] Una pila se almacena en la memoria principal y es un conjunto secuencial de posiciones a las que se hace referencia de manera similar a como ocurre con una pila física de papeles, insertando y extrayendo elementos de la cima de la misma. *Véase* el Apéndice 1B donde se incluye una explicación sobre la gestión de la pila.

REGISTROS DE CONTROL Y ESTADO

Se emplean varios registros del procesador para controlar el funcionamiento del mismo. En la mayoría de las máquinas, muchos de ellos no son visibles para el usuario. A algunos de ellos se puede acceder mediante instrucciones de máquina ejecutadas en lo que se denomina modo de control o de sistema operativo.

Por supuesto, diferentes máquinas tendrán distintas organizaciones de registros y utilizarán diferente terminología. A continuación, se proporcionará una lista razonablemente completa de tipos de registros, con una breve descripción de cada uno de ellos. Además de los registros RDIRM, RDAM, RDIE/S y RDAE/S mencionados anteriormente (Figura 1.1), los siguientes son esenciales para la ejecución de instrucciones:

- **Contador de programa (*Program Counter*, PC).** Contiene la dirección de la próxima instrucción que se leerá de la memoria.
- **Registro de instrucción (*Instruction Register*, IR).** Contiene la última instrucción leída.

Todos los diseños de procesador incluyen también un registro, o conjunto de registros, conocido usualmente como la palabra de estado del programa (*Program Status Word*, PSW), que contiene información de estado. La PSW contiene normalmente códigos de condición, además de otra información de estado, tales como un bit para habilitar/inhabilitar las interrupciones y un bit de modo usuario/supervisor.

Los **códigos de condición** (también llamados *indicadores*) son bits cuyo valor lo asigna normalmente el hardware de procesador teniendo en cuenta el resultado de las operaciones. Por ejemplo, una operación aritmética puede producir un resultado positivo, negativo, cero o desbordamiento. Además de almacenarse el resultado en sí mismo en un registro o en la memoria, se fija también un código de condición en concordancia con el resultado de la ejecución de la instrucción aritmética. Posteriormente, se puede comprobar el código de condición como parte de una operación de salto condicional. Los bits de código de condición se agrupan en uno o más registros. Normalmente, forman parte de un registro de control. Generalmente, las instrucciones de máquina permiten que estos bits se lean mediante una referencia implícita, pero no pueden ser alterados por una referencia explícita debido a que están destinados a la realimentación del resultado de la ejecución de una instrucción.

En máquinas que utilizan múltiples tipos de interrupciones, se puede proporcionar un conjunto de registros de interrupciones, con un puntero a cada rutina de tratamiento de interrupción. Si se utiliza una pila para implementar ciertas funciones (por ejemplo llamadas a procedimientos), se necesita un puntero de pila de sistema (*véase* el Apéndice 1B). El hardware de gestión de memoria, estudiado en el Capítulo 7, requiere registros dedicados. Asimismo, se pueden utilizar registros en el control de las operaciones de E/S.

En el diseño de la organización del registro de control y estado influyen varios factores. Un aspecto fundamental es proporcionar apoyo al sistema operativo. Ciertos tipos de información de control son útiles específicamente para el sistema operativo. Si el diseñador del procesador tiene un conocimiento funcional del sistema operativo que se va a utilizar, se puede diseñar la organización de registros de manera que se proporcione soporte por parte del hardware de características particulares de ese sistema operativo, en aspectos tales como la protección de memoria y la multiplexación entre programas de usuario.

Otra decisión de diseño fundamental es el reparto de la información de control entre los registros y la memoria. Es habitual dedicar las primeras (las de direcciones más bajas) cientos o miles de palabras de memoria para propósitos de control. El diseñador debe decidir cuánta información de control

debería estar en registros, más rápidos y más caros, y cuánta en la memoria principal, menos rápida y más económica.

1.3. EJECUCIÓN DE INSTRUCCIONES

Un programa que va a ejecutarse en un procesador consta de un conjunto de instrucciones almacenado en memoria. En su forma más simple, el procesamiento de una instrucción consta de dos pasos: el procesador lee (*busca*) instrucciones de la memoria, una cada vez, y ejecuta cada una de ellas. La ejecución del programa consiste en repetir el proceso de búsqueda y ejecución de instrucciones. La ejecución de la instrucción puede involucrar varias operaciones dependiendo de la naturaleza de la misma.

Se denomina *ciclo de instrucción* al procesamiento requerido por una única instrucción. En la Figura 1.2 se describe el ciclo de instrucción utilizando la descripción simplificada de dos pasos. A estos dos pasos se les denomina *fase de búsqueda* y de *ejecución*. La ejecución del programa se detiene sólo si se apaga la máquina, se produce algún tipo de error irrecuperable o se ejecuta una instrucción del programa que para el procesador.

BÚSQUEDA Y EJECUCIÓN DE UNA INSTRUCCIÓN

Al principio de cada ciclo de instrucción, el procesador lee una instrucción de la memoria. En un procesador típico, el contador del programa (PC) almacena la dirección de la siguiente instrucción que se va a leer. A menos que se le indique otra cosa, el procesador siempre incrementa el PC después de cada instrucción ejecutada, de manera que se leerá la siguiente instrucción en orden secuencial (es decir, la instrucción situada en la siguiente dirección de memoria más alta). Considere, por ejemplo, un computador simplificado en el que cada instrucción ocupa una palabra de memoria de 16 bits. Suponga que el contador del programa está situado en la posición 300. El procesador leerá la siguiente instrucción de la posición 300. En sucesivos ciclos de instrucción completados satisfactoriamente, se leerán instrucciones de las posiciones 301, 302, 303, y así sucesivamente. Esta secuencia se puede alterar, como se explicará posteriormente.

La instrucción leída se carga dentro de un registro del procesador conocido como registro de instrucción (IR). La instrucción contiene bits que especifican la acción que debe realizar el procesador. El procesador interpreta la instrucción y lleva a cabo la acción requerida. En general, estas acciones se dividen en cuatro categorías:

- **Procesador-memoria.** Se pueden transferir datos desde el procesador a la memoria o viceversa.
- **Procesador-E/S.** Se pueden enviar datos a un dispositivo periférico o recibirlos desde el mismo, transfiriéndolos entre el procesador y un módulo de E/S.

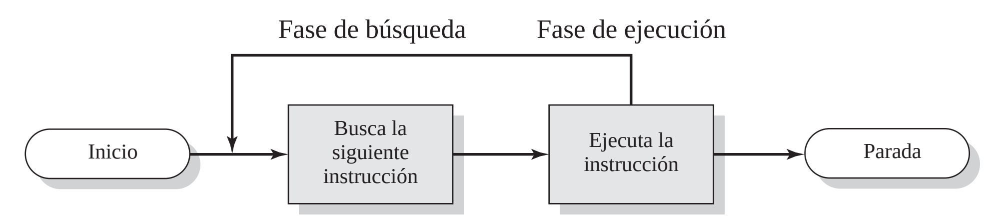


Figura 1.2. Ciclo de instrucción básico.

- **Procesamiento de datos.** El procesador puede realizar algunas operaciones aritméticas o lógicas sobre los datos.
- **Control.** Una instrucción puede especificar que se va a alterar la secuencia de ejecución. Por ejemplo, el procesador puede leer una instrucción de la posición 149, que especifica que la siguiente instrucción estará en la posición 182. El procesador almacenará en el contador del programa un valor de 182. Como consecuencia, en la siguiente fase de búsqueda, se leerá la instrucción de la posición 182 en vez de la 150.

Una ejecución de una instrucción puede involucrar una combinación de estas acciones.

Considere un ejemplo sencillo utilizando una máquina hipotética que incluye las características mostradas en la Figura 1.3. El procesador contiene un único registro de datos, llamado el acumulador (AC). Tanto las instrucciones como los datos tienen una longitud de 16 bits, estando la memoria organizada como una secuencia de palabras de 16 bits. El formato de la instrucción proporciona 4 bits para el código de operación, permitiendo hasta $2^4 = 16$ códigos de operación diferentes (representados por un único dígito hexadecimal[3]). Con los 12 bits restantes del formato de la instrucción, se pueden direccionar directamente hasta $2^{12} = 4.096$ (4K) palabras de memoria (denotadas por tres dígitos hexadecimales).

La Figura 1.4 ilustra una ejecución parcial de un programa, mostrando las partes relevantes de la memoria y de los registros del procesador. El fragmento de programa mostrado suma el contenido de la palabra de memoria en la dirección 940 al de la palabra de memoria en la dirección 941, almacenando el resultado en esta última posición. Se requieren tres instrucciones, que corresponden a tres fases de búsqueda y de ejecución, como se describe a continuación:

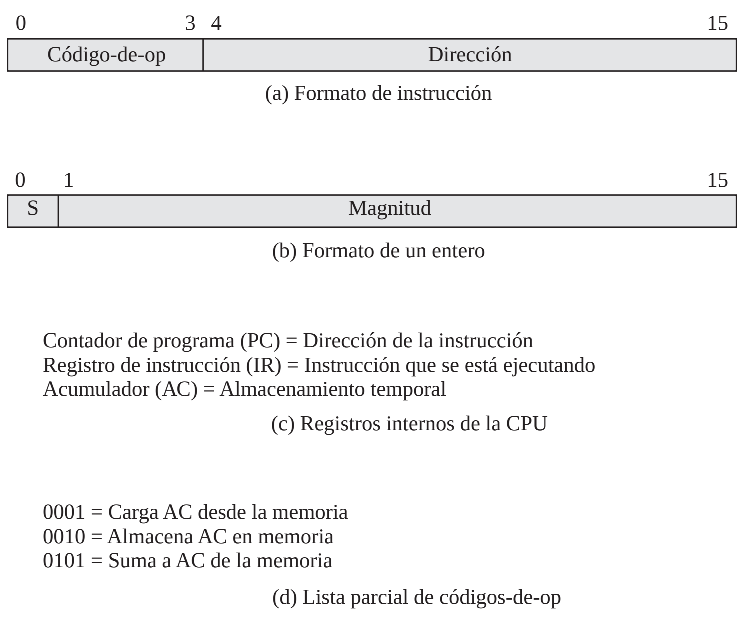


Figura 1.3. Características de una máquina hipotética.

[3] Puede encontrar un repaso básico de los sistema numéricos (decimal, binario y hexadecimal) en el *Computer Science Student Resource Site* en **WilliamStallings.com/StudentSupport.html**.

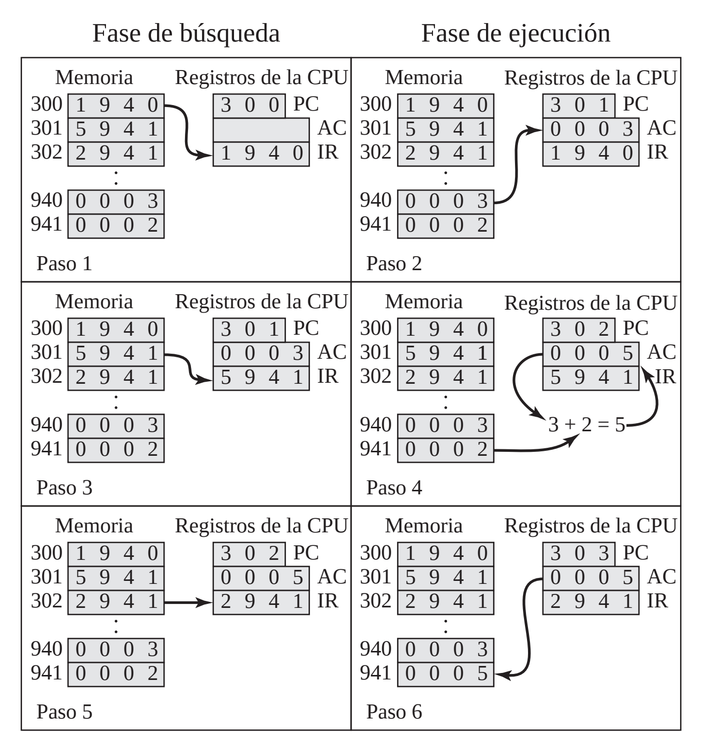


Figura 1.4. Ejemplo de ejecución de un programa (contenido de la memoria y los registros en hexadecimal).

1. El PC contiene el valor 300, la dirección de la primera instrucción. Esta instrucción (el valor 1940 en hexadecimal) se carga dentro del registro de instrucción IR y se incrementa el PC. Nótese que este proceso involucra el uso del registro de dirección de memoria (RDIM) y el registro de datos de memoria (RDAM). Para simplificar, no se muestran estos registros intermedios.
2. Los primeros 4 bits (primer dígito hexadecimal) en el IR indican que en el AC se va a cargar un valor leído de la memoria. Los restantes 12 bits (tres dígitos hexadecimales) especifican la dirección de memoria, que es 940.
3. Se lee la siguiente instrucción (5941) de la posición 301 y se incrementa el PC.
4. El contenido previo del AC y el contenido de la posición 941 se suman y el resultado se almacena en el AC.
5. Se lee la siguiente instrucción (2941) de la posición 302 y se incrementa el PC.
6. Se almacena el contenido del AC en la posición 941.

En este ejemplo, se necesitan tres ciclos de instrucción, de tal forma que cada uno consta de una fase de búsqueda y una fase de ejecución, para sumar el contenido de la posición 940 al contenido de la 941. Con un juego de instrucciones más complejo, se necesitarían menos ciclos de instrucción. La mayoría de los procesadores modernos incluyen instrucciones que contienen más de una dirección. Por tanto, la fase de ejecución de una determinada instrucción puede involucrar más de una referencia a memoria. Asimismo, en vez de referencias a memoria, una instrucción puede especificar una operación de E/S.

SISTEMA DE E/S

Se pueden intercambiar datos directamente entre un módulo de E/S (por ejemplo, un controlador de disco) y el procesador. Al igual que el procesador puede iniciar una lectura o una escritura en memoria, especificando la dirección de una posición de memoria, también puede leer o escribir datos en un módulo de E/S. En este caso, el procesador identifica un dispositivo específico que está controlado por un determinado módulo de E/S. Por tanto, podría producirse una secuencia de instrucciones similar a la de la Figura 1.4, con instrucciones de E/S en vez de instrucciones que hacen referencia a memoria.

En algunos casos, es deseable permitir que los intercambios de E/S se produzcan directamente con la memoria para liberar al procesador de la tarea de E/S. En tales casos, el procesador concede a un módulo de E/S la autorización para leer o escribir de la memoria, de manera que la transferencia entre memoria y E/S puede llevarse a cabo sin implicar al procesador. Durante dicha transferencia, el módulo de E/S emite mandatos de lectura y escritura a la memoria, liberando al procesador de la responsabilidad del intercambio. Esta operación, conocida como acceso directo a memoria (*Direct Memory Access*, DMA) se examinará al final de este capítulo.

1.4. INTERRUPCIONES

Prácticamente todos los computadores proporcionan un mecanismo por el cual otros módulos (memoria y E/S) pueden interrumpir el secuenciamiento normal del procesador. La Tabla 1.1 detalla los tipos más comunes de interrupciones.

Básicamente, las interrupciones constituyen una manera de mejorar la utilización del procesador. Por ejemplo, la mayoría de los dispositivos de E/S son mucho más lentos que el procesador. Supóngase que el procesador está transfiriendo datos a una impresora utilizando el esquema de ciclo de instrucción de la Figura 1.2. Después de cada instrucción de escritura, el procesador debe parar y permanecer inactivo hasta que la impresora la lleve a cabo. La longitud de esta pausa puede ser del orden de muchos miles o incluso millones de ciclos de instrucción. Claramente, es un enorme desperdicio de la capacidad del procesador.

Tabla 1.1. Clases de interrupciones.

De programa	Generada por alguna condición que se produce como resultado de la ejecución de una instrucción, tales como un desbordamiento aritmético, una división por cero, un intento de ejecutar una instrucción de máquina ilegal, y las referencias fuera del espacio de la memoria permitido para un usuario.
Por temporizador	Generada por un temporizador del procesador. Permite al sistema operativo realizar ciertas funciones de forma regular.
De E/S	Generada por un controlador de E/S para señalar la conclusión normal de una operación o para indicar diversas condiciones de error.
Por fallo del hardware	Generada por un fallo, como un fallo en el suministro de energía o un error de paridad en la memoria.

Para dar un ejemplo concreto, considere un computador personal que operase a 1GHz, lo que le permitiría ejecutar aproximadamente 10^9 instrucciones por segundo[4]. Un típico disco duro tiene una

[4] Una discussion de los usos de prefijos numéricos, tales como giga y tera, está disponible en un documento de apoyo en el *Computer Science Student Resource Site* en **WilliamStallings.com/StudentSupport.html.**

velocidad de rotación de 7200 revoluciones por minuto, que corresponde con un tiempo de rotación de media pista de 4 ms., que es 4 millones de veces más lento que el procesador.

La Figura 1.5a muestra esta cuestión. El programa de usuario realiza una serie de llamadas de ESCRITURA intercaladas con el procesamiento. Los segmentos de código 1, 2 y 3 se refieren a secuencias de instrucciones que no involucran E/S. Las llamadas de ESCRITURA invocan a una rutina de E/S que es una utilidad del sistema que realizará la operación real de E/S. El programa de E/S consta de tres secciones:

- Una secuencia de instrucciones, etiquetada como 4 en la figura, para preparar la operación real de E/S. Esto puede incluir copiar los datos de salida en un *buffer* especial y preparar los parámetros de un mandato para el dispositivo.
- El mandato real de E/S. Sin el uso de interrupciones, una vez que se emite este mandato, el programa debe esperar a que el dispositivo de E/S realice la función solicitada (o comprobar periódicamente el estado, o muestrear, el dispositivo de E/S). El programa podría esperar simplemente realizando repetidamente una operación de comprobación para determinar si se ha realizado la operación de E/S.
- Una secuencia de instrucciones, etiquetada como 5 en la figura, para completar la operación. Esto puede incluir establecer un valor que indique el éxito o el fallo de la operación.

Debido a que la operación de E/S puede tardar un tiempo relativamente largo hasta que se completa, el programa de E/S se queda colgado esperando que se complete; por ello, el programa de usuario se detiene en el momento de la llamada de ESCRITURA durante un periodo de tiempo considerable.

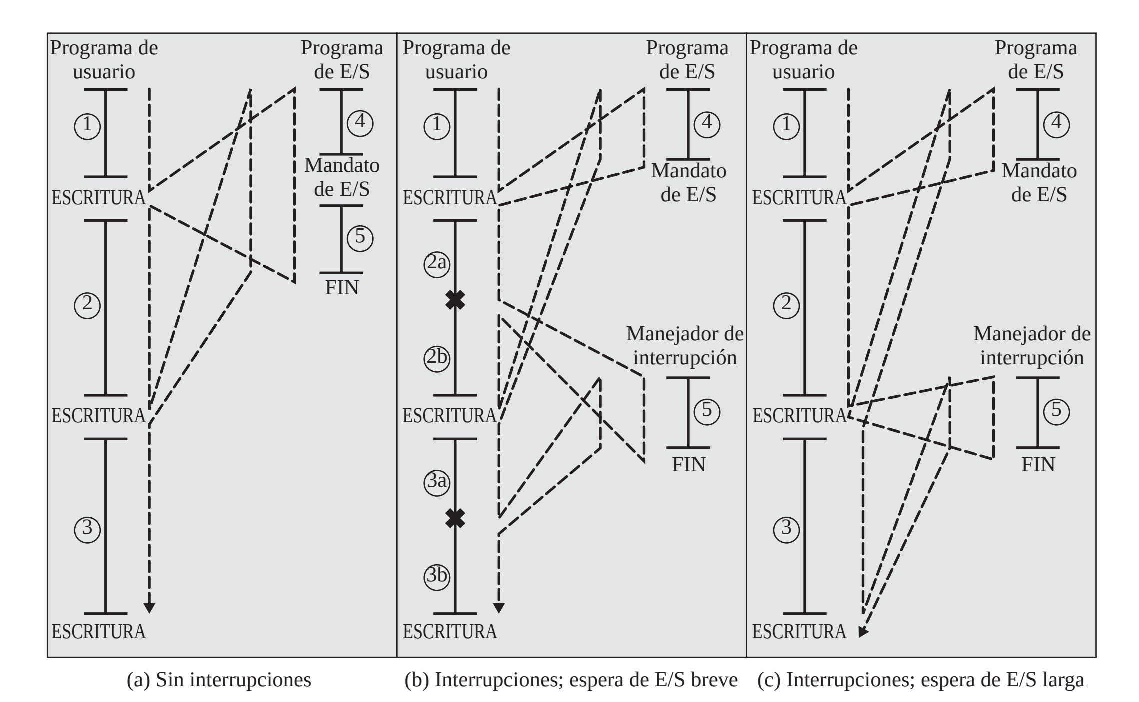


Figura 1.5. Flujo de programa del control sin interrupciones y con ellas.

INTERRUPCIONES Y EL CICLO DE INSTRUCCIÓN

Gracias a las interrupciones, el procesador puede dedicarse a ejecutar otras instrucciones mientras que la operación de E/S se está llevando a cabo. Considere el flujo de control mostrado en la Figura 1.5b. Como anteriormente, el programa de usuario alcanza un punto en el que hace una llamada al sistema que consiste en una llamada de ESCRITURA. El programa de E/S que se invoca en este caso consta sólo del código de preparación y el mandato real de E/S. Después de que se ejecuten estas pocas instrucciones, se devuelve el control al programa de usuario. Mientras tanto, el dispositivo externo está ocupado aceptando datos de la memoria del computador e imprimiéndolos. La operación de E/S se lleva a cabo de forma concurrente con la ejecución de instrucciones en el programa de usuario.

Cuando el dispositivo externo está listo para ser atendido, es decir, cuando está preparado para aceptar más datos del procesador, el módulo de E/S de este dispositivo externo manda una señal de *petición de interrupción* al procesador. El procesador responde suspendiendo la ejecución del programa actual, saltando a la rutina de servicio específica de este dispositivo de E/S, conocida como manejador de interrupción, y reanudando la ejecución original después de haber atendido al dispositivo. En la Figura 1.5b se indican con una **X** los puntos en los que se produce cada interrupción. Téngase en cuenta que se puede producir una interrupción en cualquier punto de la ejecución del programa principal, no sólo en una determinada instrucción.

De cara al programa de usuario, una interrupción suspende la secuencia normal de ejecución. Cuando se completa el procesamiento de la interrupción, se reanuda la ejecución (Figura 1.6). Por tanto, el programa de usuario no tiene que contener ningún código especial para tratar las interrupciones; el procesador y el sistema operativo son responsables de suspender el programa de usuario y, posteriormente, reanudarlo en el mismo punto.

Para tratar las *interrupciones*, se añade una *fase de interrupción* al ciclo de instrucción, como se muestra en la Figura 1.7 (compárese con la Figura 1.2). En la fase de interrupción, el procesador comprueba si se ha producido cualquier interrupción, hecho indicado por la presencia de una señal de interrupción. Si no hay interrupciones pendientes, el procesador continúa con la fase de búsqueda y lee la siguiente instrucción del programa actual. Si está pendiente una interrupción, el procesador suspende la ejecución del programa actual y ejecuta la rutina del *manejador de interrupción*. La rutina del manejador de interrupción es generalmente parte del sistema operativo. Normalmente, esta rutina

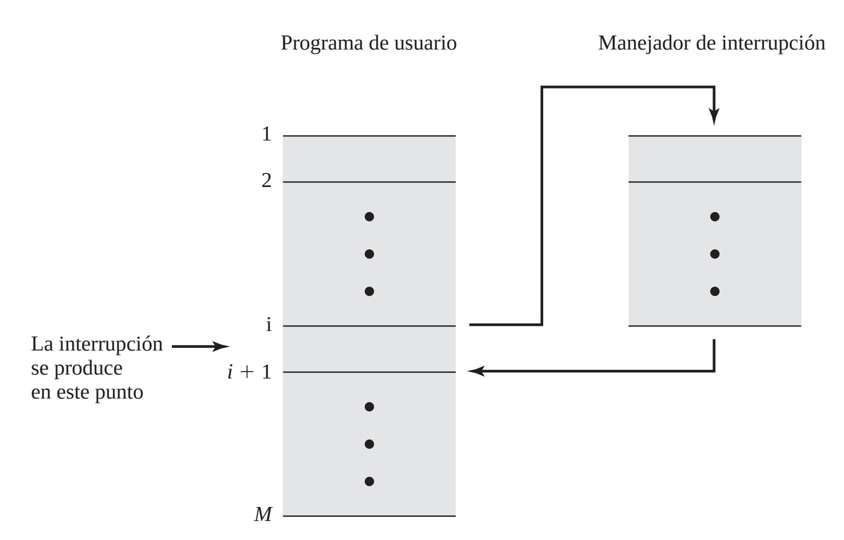


Figura 1.6. Transferencia de control mediante interrupciones.

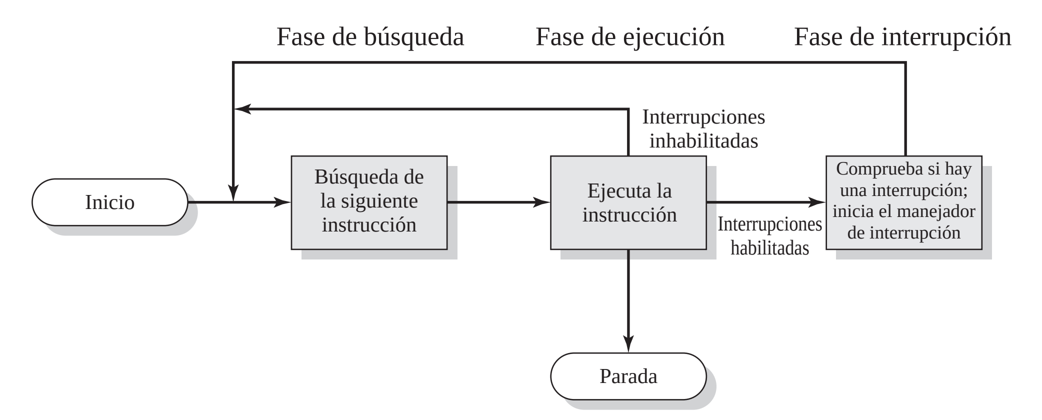


Figura 1.7. Ciclo de instrucción con interrupciones.

determina la naturaleza de la interrupción y realiza las acciones que se requieran. En el ejemplo que se está usando, el manejador determina qué módulo de E/S generó la interrupción y puede dar paso a un programa que escriba más datos en ese módulo de E/S. Cuando se completa la rutina del manejador de interrupción, el procesador puede reanudar la ejecución del programa de usuario en el punto de la interrupción.

Es evidente que este proceso implica cierta sobrecarga. Deben ejecutarse instrucciones adicionales (en el manejador de interrupción) para determinar la naturaleza de la interrupción y decidir sobre la acción apropiada. Sin embargo, debido a la cantidad relativamente elevada de tiempo que se gastaría simplemente a la espera de una operación de E/S, el procesador se puede emplear mucho más eficientemente con el uso de interrupciones.

Para apreciar la ganancia en eficiencia, considere la Figura 1.8, que es un diagrama de tiempo basado en el flujo de control de las Figuras 1.5a y 1.5b. Las Figuras 1.5b y 1.8 asumen que el tiempo requerido para la operación de E/S es relativamente corto: inferior al tiempo que tarda en completarse la ejecución de instrucciones entre las operaciones de escritura del programa de usuario. El caso más típico, especialmente para un dispositivo lento como una impresora, es que la operación de E/S tarde mucho más tiempo que la ejecución de una secuencia de instrucciones de usuario. La Figura 1.5c ilustra este tipo de situación. En este caso, el programa de usuario alcanza la segunda llamada de ESCRITURA antes de que se complete la operación de E/S generada por la primera llamada. El resultado es que el programa de usuario se queda colgado en ese punto. Cuando se completa la operación de E/S precedente, se puede procesar la nueva llamada de ESCRITURA y se puede empezar una nueva operación de E/S. La Figura 1.9 muestra la temporización de esta situación con el uso de interrupciones o sin ellas. Se puede observar que hay una ganancia en eficiencia debido a que parte del tiempo durante el que se realiza la operación de E/S se solapa con la ejecución de las instrucciones del usuario.

PROCESAMIENTO DE INTERRUPCIONES

La aparición de una interrupción dispara varios eventos, tanto en el hardware del procesador como en el software. La Figura 1.10 muestra una secuencia típica. Cuando un dispositivo de E/S completa una operación de E/S, se produce la siguiente secuencia de eventos en el hardware:

1. El dispositivo genera una señal de interrupción hacia el procesador.
2. El procesador termina la ejecución de la instrucción actual antes de responder a la interrupción, como se indica en la Figura 1.7.

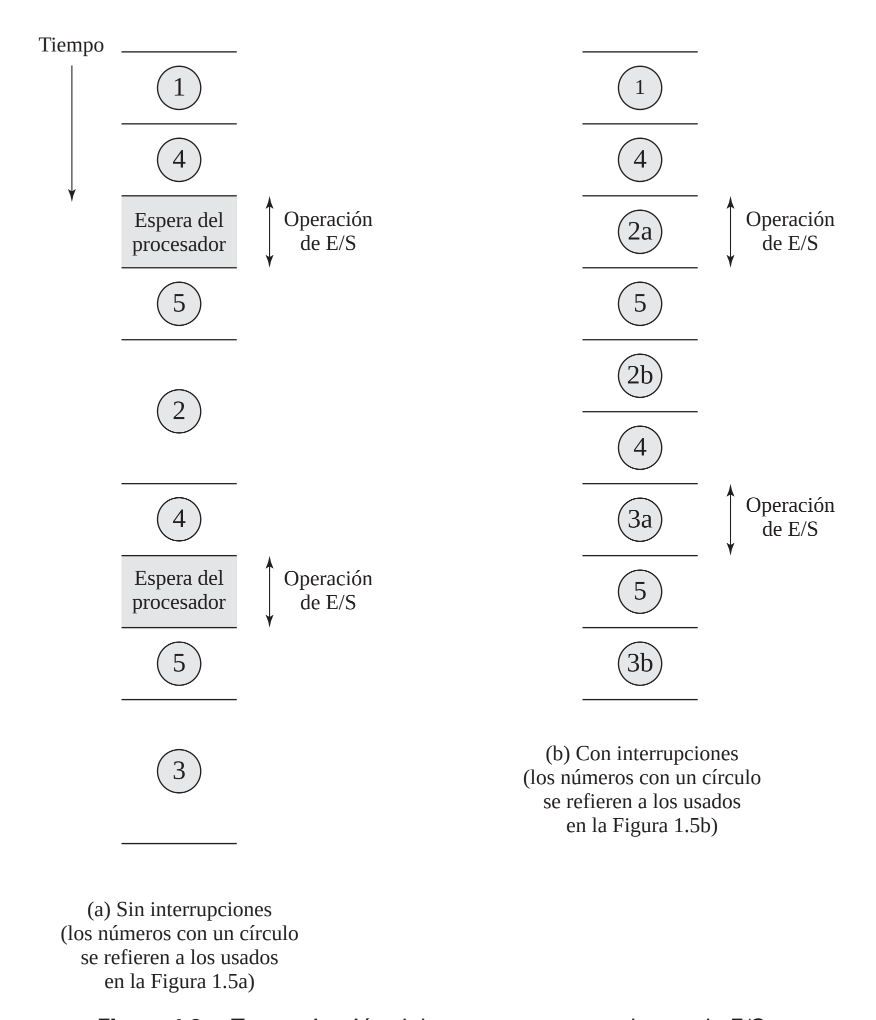


Figura 1.8. Temporización del programa: espera breve de E/S.

3. El procesador comprueba si hay una petición de interrupción pendiente, determina que hay una y manda una señal de reconocimiento al dispositivo que produjo la interrupción. Este reconocimiento permite que el dispositivo elimine su señal de interrupción.

4. En ese momento, el procesador necesita prepararse para transferir el control a la rutina de interrupción. Para comenzar, necesita salvar la información requerida para reanudar el programa actual en el momento de la interrupción. La información mínima requerida es la palabra de estado del programa (PSW) y la posición de la siguiente instrucción que se va a ejecutar, que está contenida en el contador de programa. Esta información se puede apilar en la pila de control de sistema (*véase* el Apéndice 1B).

5. A continuación, el procesador carga el contador del programa con la posición del punto de entrada de la rutina de manejo de interrupción que responderá a esta interrupción. Dependiendo de la arquitectura de computador y del diseño del sistema operativo, puede haber un único programa, uno por cada tipo de interrupción o uno por cada dispositivo y tipo de interrupción. Si hay más de una rutina de manejo de interrupción, el procesador debe determinar cuál invocar. Esta información puede estar incluida en la señal de interrupción original o el procesador

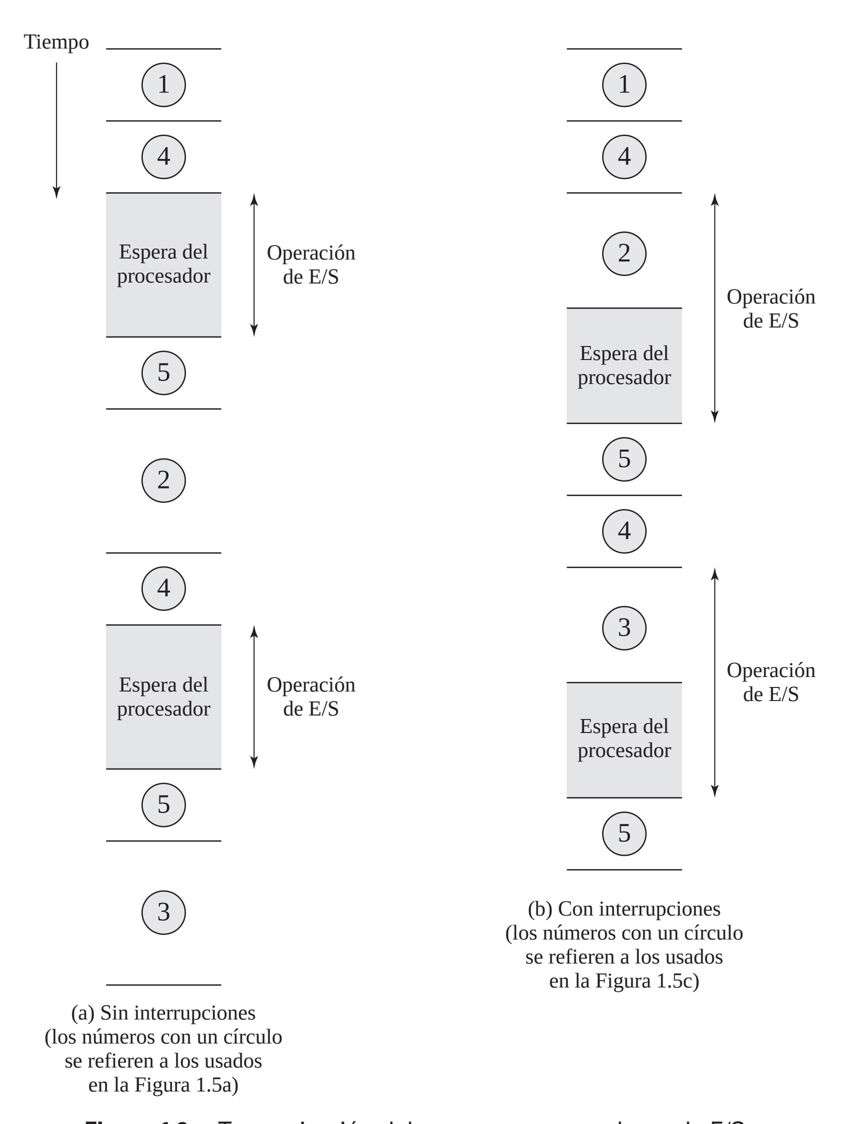


Figura 1.9. Temporización del programa: espera larga de E/S.

puede tener que realizar una petición al dispositivo que generó la interrupción para obtener una respuesta que contiene la información requerida.

Una vez que se ha cargado el contador del programa, el procesador continúa con el siguiente ciclo de instrucción, que comienza con una lectura de instrucción. Dado que la lectura de la instrucción está determinada por el contenido del contador del programa, el resultado es que se transfiere el control al programa manejador de interrupción. La ejecución de este programa conlleva las siguientes operaciones:

6. En este momento, el contador del programa y la PSW vinculados con el programa interrumpido se han almacenado en la pila del sistema. Sin embargo, hay otra información que se considera parte del estado del programa en ejecución. En concreto, se necesita salvar el contenido de los registros del procesador, puesto que estos registros los podría utilizar el manejador de interrup-

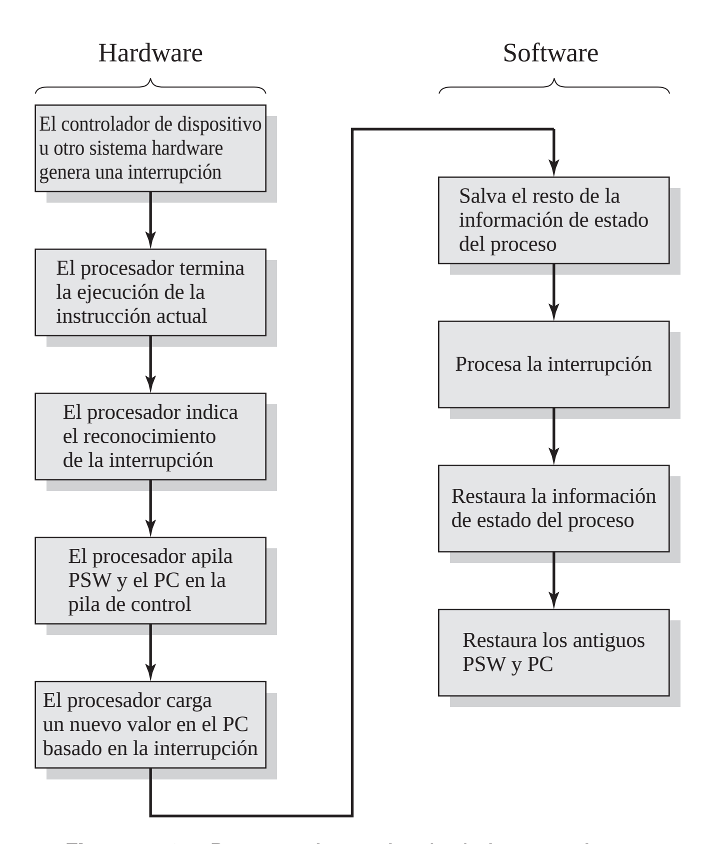


Figura 1.10. Procesamiento simple de interrupciones.

ciones. Por tanto, se deben salvar todos estos valores, así como cualquier otra información de estado. Generalmente, el manejador de interrupción comenzará salvando el contenido de todos los registros en la pila. En el Capítulo 3 se estudiará qué otra información de estado debe salvarse. La Figura 1.11a muestra un ejemplo sencillo. En este caso, un programa de usuario se interrumpe después de la instrucción en la posición N. El contenido de todos los registros, así como la dirección de la siguiente instrucción ($N + 1$), un total de M palabras, se apilan en la pila de control. El puntero de pila se actualiza para que señale a la nueva cima de la pila, mientras que el contador de programa quedará apuntando al principio de la rutina de servicio de interrupción.

7. El manejador de interrupción puede en este momento comenzar a procesar la interrupción. Esto incluirá un examen de la información de estado relacionada con la operación de E/S o con otro evento distinto que haya causado la interrupción. Asimismo, puede implicar el envío de mandatos adicionales o reconocimientos al dispositivo de E/S.

8. Cuando se completa el procesamiento de la interrupción, se recuperan los valores de los registros salvados en la pila y se restituyen en los registros (como ejemplo, *véase* la Figura 1.11b).

9. La última acción consiste en restituir de la pila los valores de la PSW y del contador del programa. Como resultado, la siguiente instrucción que se va ejecutar corresponderá al programa previamente interrumpido.

Es importante salvar toda la información de estado del programa interrumpido para su posterior reanudación. Esto se debe a que la interrupción no es una rutina llamada desde el programa. En su lu-

gar, la interrupción puede suceder en cualquier momento y, por tanto, en cualquier punto de la ejecución de un programa de usuario. Su aparición es imprevisible.

MÚLTIPLES INTERRUPCIONES

El estudio realizado hasta el momento ha tratado solamente el caso de que se produzca una única interrupción. Supóngase, sin embargo, que se producen múltiples interrupciones. Por ejemplo, un programa puede estar recibiendo datos de una línea de comunicación e imprimiendo resultados al mismo tiempo. La impresora generará una interrupción cada vez que completa una operación de impresión. El controlador de la línea de comunicación generará una interrupción cada vez que llega una unidad de datos. La unidad podría consistir en un único carácter o en un bloque, dependiendo de la naturaleza del protocolo de comunicaciones. En cualquier caso, es posible que se produzca una interrupción de comunicación mientras se está procesando una interrupción de la impresora.

Se pueden considerar dos alternativas a la hora de tratar con múltiples interrupciones. La primera es inhabilitar las interrupciones mientras que se está procesando una interrupción. Una *interrupción inhabilitada* significa simplemente que el procesador ignorará cualquier nueva señal de petición de interrupción. Si se produce una interrupción durante este tiempo, generalmente permanecerá pendiente de ser procesada, de manera que el procesador sólo la comprobará después de que se rehabiliten las interrupciones. Por tanto, cuando se ejecuta un programa de usuario y se produce una interrupción, se inhabilitan las interrupciones inmediatamente. Después de que se completa la rutina de manejo de la interrupción, se rehabilitan las interrupciones antes de reanudar el programa de usuario, y el procesador comprueba si se han producido interrupciones adicionales. Esta estrategia es válida y sencilla, puesto que las interrupciones se manejan en estricto orden secuencial (Figura 1.12a).

La desventaja de la estrategia anterior es que no tiene en cuenta la prioridad relativa o el grado de urgencia de las interrupciones. Por ejemplo, cuando llegan datos por la línea de comunicación, se puede necesitar que se procesen rápidamente de manera que se deje sitio para otros datos que pueden llegar. Si el primer lote de datos no se ha procesado antes de que llegue el segundo, los datos pueden perderse porque el *buffer* del dispositivo de E/S puede llenarse y desbordarse.

Una segunda estrategia es definir prioridades para las interrupciones y permitir que una interrupción de más prioridad cause que se interrumpa la ejecución de un manejador de una interrupción de menor prioridad (Figura 1.12b). Como ejemplo de esta segunda estrategia, considere un sistema con tres dispositivos de E/S: una impresora, un disco y una línea de comunicación, con prioridades crecientes de 2, 4 y 5, respectivamente. La Figura 1.13, basada en un ejemplo de [TANE97], muestra una posible secuencia. Un programa de usuario comienza en $t = 0$. En $t = 10$, se produce una interrupción de impresora; se almacena la información de usuario en la pila del sistema y la ejecución continúa en la rutina de servicio de interrupción (*Interrupt Service Routine*, ISR) de la impresora. Mientras todavía se está ejecutando esta rutina, en $t = 15$ se produce una interrupción del equipo de comunicaciones. Debido a que la línea de comunicación tiene una prioridad superior a la de la impresora, se sirve la petición de interrupción. Se interrumpe la ISR de la impresora, se almacena su estado en la pila y la ejecución continúa con la ISR del equipo de comunicaciones. Mientras se está ejecutando esta rutina, se produce una interrupción del disco ($t = 20$). Dado que esta interrupción es de menor prioridad, simplemente se queda en espera, y la ISR de la línea de comunicación se ejecuta hasta su conclusión.

Cuando se completa la ISR de la línea de comunicación ($t = 25$), se restituye el estado previo del proceso, que corresponde con la ejecución de la ISR de la impresora. Sin embargo, antes incluso de que pueda ejecutarse una sola instrucción de esta rutina, el procesador atiende la interrupción de disco de mayor prioridad y transfiere el control a la ISR del disco. Sólo cuando se completa esa rutina ($t = 35$), se reanuda la ISR de la impresora. Cuando esta última rutina se completa ($t = 40$), se devuelve finalmente el control al programa de usuario.

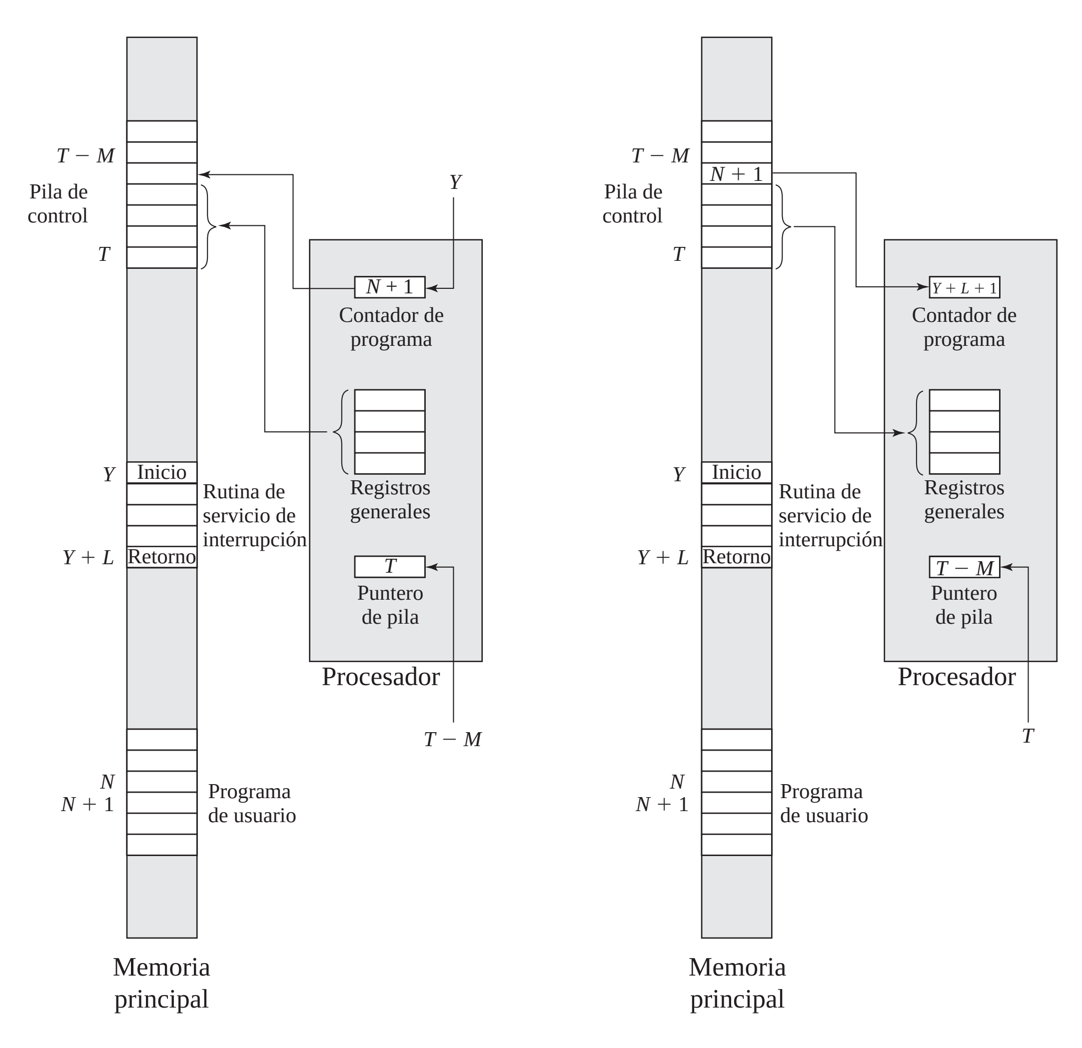
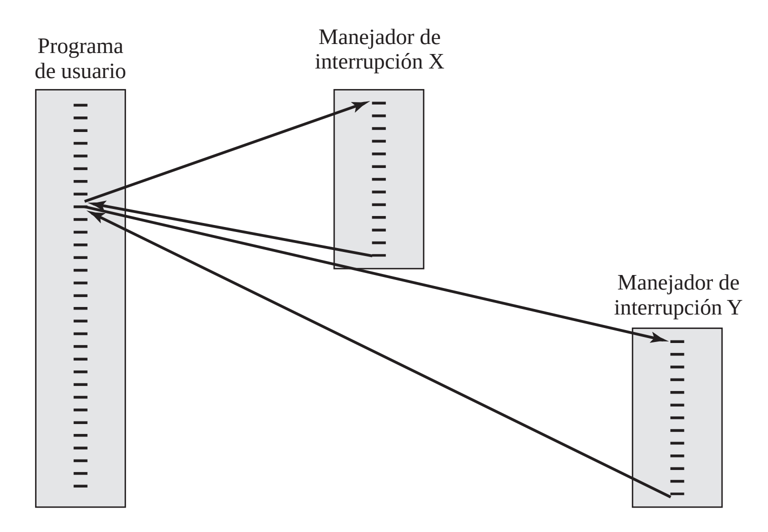


Figura 1.11. Cambios en la memoria y en los registros durante una interrupción.

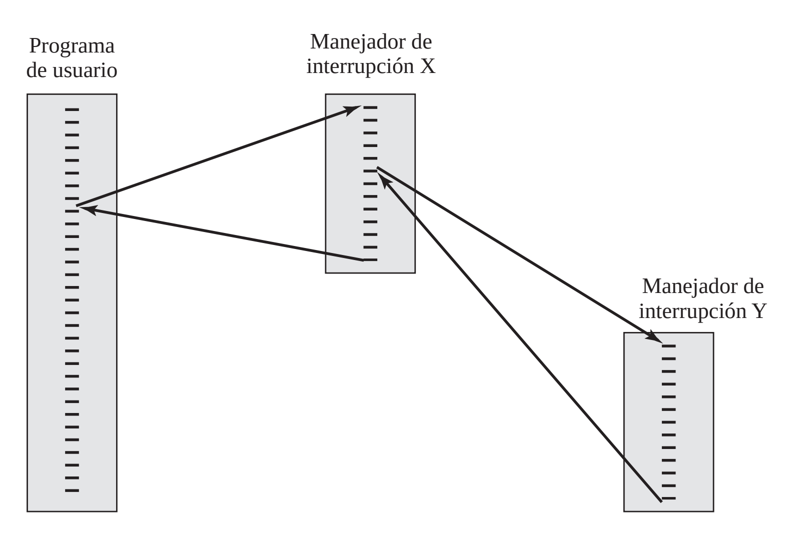
MULTIPROGRAMACIÓN

Incluso utilizando interrupciones, puede que el procesador siga sin utilizarse eficientemente. Por ejemplo, considérese la Figura 1.9b, que demuestra una mejor utilización del procesador. Si el tiempo requerido para completar una operación de E/S es mucho mayor que el código de usuario entre las llamadas de E/S (una situación habitual), el procesador estará parado la mayor parte del tiempo. Una solución a este problema es permitir que múltiples programas de usuario estén activos al mismo tiempo.

Supóngase, por ejemplo, que el procesador tiene que ejecutar dos programas. Uno de ellos simplemente se dedica a leer datos de la memoria y copiarlos a un dispositivo externo; el otro es algún tipo de aplicación que implica mucho cálculo. El procesador puede empezar con el programa que



(a) Procesamiento secuencial de interrupciones



(b) Procesamiento anidado de interrupciones

Figura 1.12. Transferencia de control con múltiples interrupciones.

genera salida, emitir un mandato de escritura al dispositivo externo y, a continuación, empezar la ejecución de la otra aplicación. Cuando el procesador trata con varios programas, la secuencia en la que se ejecutan los programas dependerá de su prioridad relativa, así como de si están esperando la finalización de una operación de E/S. Cuando se interrumpe un programa y se transfiere el control a un manejador de interrupción, una vez que se ha completado la rutina del manejador de interrupción, puede ocurrir que no se le devuelva inmediatamente el control al programa de usuario que estaba en ejecución en ese momento. En su lugar, el control puede pasar a algún otro programa pendiente de ejecutar que tenga una prioridad mayor. Posteriormente, se reanudará el programa de usuario interrumpido previamente, en el momento en que tenga la mayor prioridad. Este concepto de múltiples programas que ejecutan en turnos se denomina multiprogramación y se estudiará más adelante en el Capítulo 2.

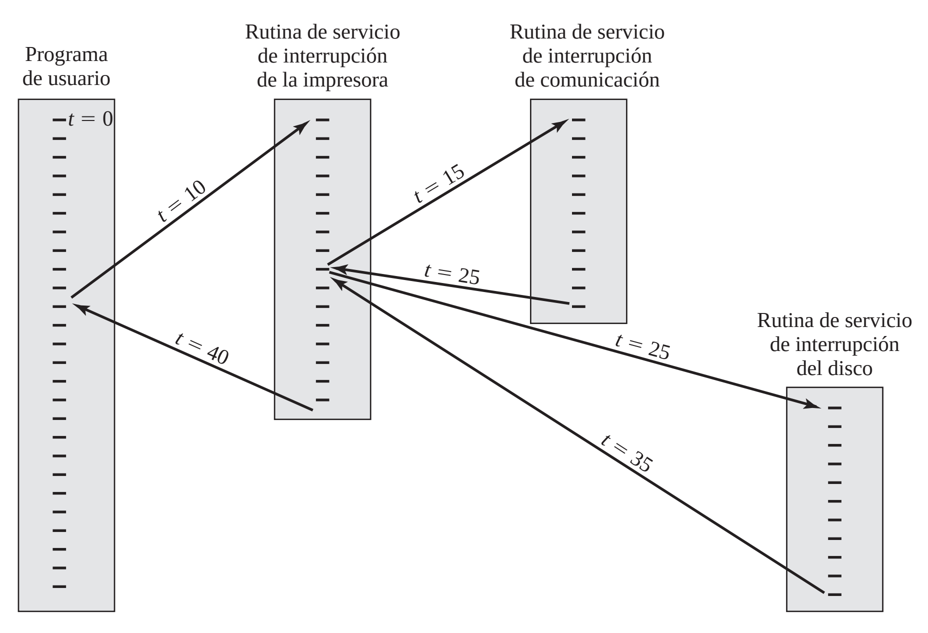


Figura 1.13. Ejemplo de secuencia de tiempo con múltiples interrupciones.

1.5. LA JERARQUÍA DE MEMORIA

Las restricciones de diseño en la memoria de un computador se pueden resumir en tres preguntas: ¿cuál es su capacidad? ¿Cuál es su velocidad? ¿Cuál es su coste?

La pregunta sobre cuánta debe ser su capacidad es algo que no tiene límite. Si se dispone de una determinada capacidad, probablemente se desarrollarán aplicaciones que la usarán. La cuestión acerca de la velocidad tiene, hasta cierto tiempo, una respuesta más fácil. Para alcanzar un rendimiento máximo, la memoria debe ser capaz de mantener el ritmo del procesador. Es decir, según el procesador va ejecutando instrucciones, no debería haber pausas esperando que estén disponibles las instrucciones o los operandos. Se debe considerar también la última pregunta. Para un sistema práctico, el coste de la memoria debe ser razonable en relación con los otros componentes.

Como se podría esperar, hay un compromiso entre las tres características fundamentales de la memoria: a saber, coste, capacidad y tiempo de acceso. En cualquier momento dado, se utilizan diversas tecnologías para implementar los sistemas de memoria. En todo este espectro de tecnologías, se cumplen las siguientes relaciones:

- Cuanto menor tiempo de acceso, mayor coste por bit.
- Cuanto mayor capacidad, menor coste por bit.
- Cuanto mayor capacidad, menor velocidad de acceso.

Queda claro el dilema al que se enfrenta el diseñador. A él le gustaría utilizar tecnologías que proporcionen una memoria de gran capacidad, tanto porque se necesita esa capacidad como porque su coste por bit es bajo. Sin embargo, para cumplir con los requisitos de rendimiento, el diseñador necesita utilizar memorias de capacidad relativamente baja con tiempos de acceso rápidos.

La solución a este dilema consiste en no basarse en un único componente de memoria o en una sola tecnología, sino emplear una **jerarquía de memoria**. En la Figura 1.14 se muestra una jerarquía típica. Según se desciende en la jerarquía, ocurre lo siguiente:

a) Disminución del coste por bit.

b) Aumento de la capacidad.

c) Aumento del tiempo de acceso.

d) Disminución de la frecuencia de acceso a la memoria por parte del procesador.

Por tanto, las memorias más rápidas, caras y pequeñas se complementan con memorias más lentas, baratas y grandes. La clave para el éxito de esta organización es el último aspecto: la disminución de la frecuencia de acceso. Este concepto se examinará con mayor detalle más adelante en este mismo capítulo, cuando se estudie la cache, y en posteriores capítulos del libro, cuando se presente la memoria virtual. De todos modos, se proporcionará una breve explicación en ese momento.

Supóngase que el procesador tiene acceso a dos niveles de memoria. El nivel 1 contiene 1.000 bytes y tiene un tiempo de acceso de 0.1 µs; el nivel 2 contiene 100.000 bytes y tiene un tiempo de acceso de 1µs. Asuma que si un byte que se va a acceder está en el nivel 1, el procesador lo hace di-

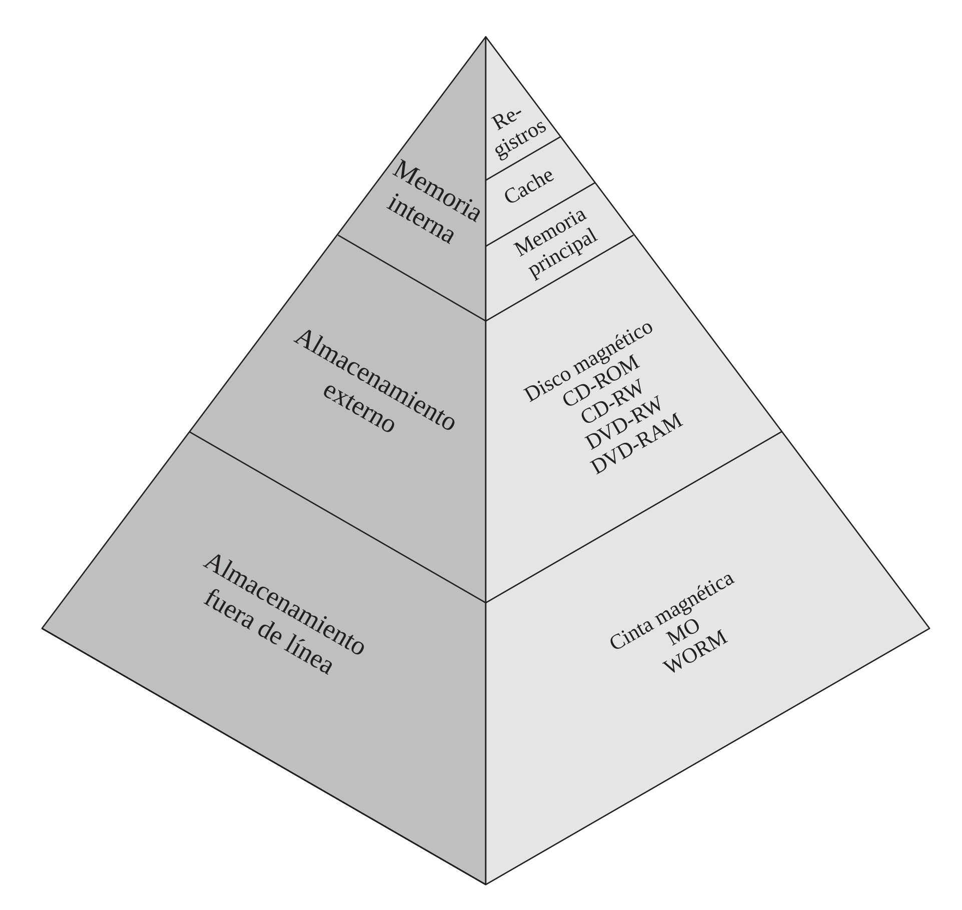


Figura 1.14. La jerarquía de memoria.

rectamente. Sin embargo, si está en el nivel 2, primero se transfiere el byte al nivel 1 y, a continuación, el procesador lo accede. Para simplificar, se ignorará el tiempo requerido por el procesador para determinar si el byte está en el nivel 1 o en el 2. La Figura 1.15 muestra la forma general de la curva que representa esta situación. La figura muestra el tiempo de acceso medio a una memoria de dos niveles como una función de la **tasa de aciertos** *A*, donde *A* se define como la fracción de todos los accesos a memoria que se encuentran en la memoria más rápida (por ejemplo, la cache), T_1 es el tiempo de acceso al nivel 1 y T_2 al nivel 2[5]. Como se puede observar, para porcentajes elevados de accesos al nivel 1, el tiempo medio total de acceso está mucho más próximo al correspondiente al nivel 1 que al del nivel 2.

En el ejemplo, se supone que el 95% de los accesos a memoria se encuentran en la cache (A = 0,95). Por tanto, el tiempo medio para acceder a un byte se puede expresar como:

$$(0{,}95)(0{,}1\ \mu s) + (0{,}05)(0{,}1\ \mu s + 1\ \mu s) = 0{,}095 + 0{,}055 = 0{,}15\ \mu s$$

El resultado está próximo al tiempo de acceso de la memoria más rápida. Por tanto, en principio, la estrategia de utilizar dos niveles de memoria funciona, pero sólo si se cumplen las condiciones de la (a) a la (d). Mediante el empleo de diversas tecnologías, existe un rango de sistemas de memoria que satisfacen las condiciones de la (a) a la (c). Afortunadamente, la condición (d) también es generalmente válida.

La validez de la condición (d) está basada en un principio conocido como la *proximidad de referencias* [DENN68]. Durante el curso de ejecución de un programa, las referencias de memoria del procesador, tanto a instrucciones como a datos, tienden a agruparse. Los programas contienen habitualmente diversos bucles iterativos y subrutinas. Una vez que se inicia un bucle o una subrutina, hay referencias repetidas a un pequeño conjunto de instrucciones. Del mismo modo, las operaciones con tablas y vectores involucran accesos a conjuntos agrupados de bytes de datos. En un periodo de tiem-

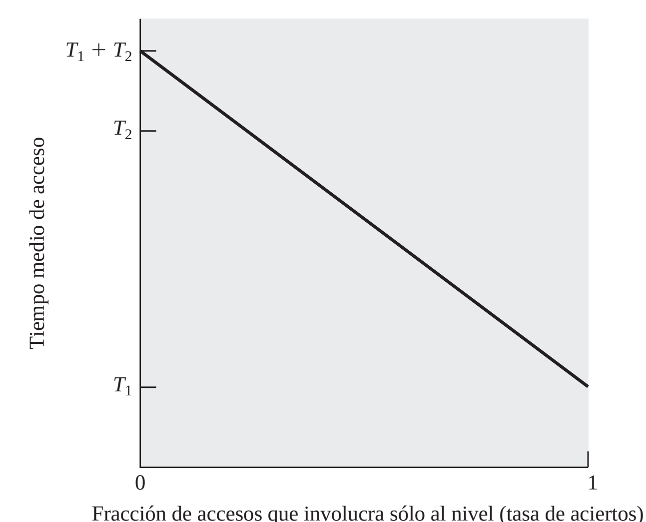


Figura 1.15. Rendimiento de una memoria simple de dos niveles.

[5] Si la palabra accedida se encuentra en la memoria más rápida, a esto se le define como un **acierto**, mientras que un **fallo** sucede cuando la palabra accedida no está en esta memoria más rápida.

po largo, las agrupaciones que se están usando van cambiando, pero en un periodo corto, el procesador está principalmente trabajando con grupos fijos de referencias a memoria.

Por consiguiente, es posible organizar los datos a través de la jerarquía de manera que el porcentaje de accesos a cada nivel sucesivamente más bajo es considerablemente menor que al nivel inferior. Considere el ejemplo de dos niveles presentado previamente. Supóngase que la memoria de nivel 2 contiene todos los datos e instrucciones del programa. Las agrupaciones actuales se pueden almacenar temporalmente en el nivel 1. De vez en cuando, una de las agrupaciones en el nivel 1 tendrá que ser expulsada al nivel 2 para hacer sitio a una nueva agrupación que llega al nivel 1. De media, sin embargo, la mayoría de las referencias corresponderá con instrucciones y datos contenidos en el nivel 1.

Este principio se puede aplicar a más de dos niveles de memoria. El tipo de memoria más rápida, pequeña y costosa consiste en los registros internos del procesador. Normalmente, un procesador contendrá unas pocas docenas de ese tipo de registros, aunque algunas máquinas contienen cientos de registros. Descendiendo dos niveles, está la memoria principal que es el sistema de memoria interna fundamental del computador. Cada posición de memoria principal tiene una única dirección. La mayoría de las instrucciones de máquina hacen referencia a una o más direcciones de memoria principal. La memoria principal se amplía usualmente con una cache, más pequeña y de mayor velocidad. La cache normalmente no es visible al programador o, incluso, al procesador. Se trata de un dispositivo que controla el movimiento de datos entre la memoria principal y los registros de procesador con objeto de mejorar el rendimiento.

Las tres formas de memoria descritas son, normalmente, volátiles y emplean tecnología de semiconductores. El uso de tres niveles explota el hecho de que la memoria de semiconductores se presenta en diversos tipos, que se diferencian en velocidad y coste. Los datos se almacenan de forma más permanente en los dispositivos de almacenamiento masivo externos, de los cuales los más comunes son los discos duros y los dispositivos extraíbles, tales como los discos extraíbles, las cintas y el almacenamiento óptico. La memoria no volátil externa se denomina también **memoria secundaria** o **memoria auxiliar**. Se usa para almacenar los ficheros de programas y datos, siendo usualmente visible al programador sólo en términos de ficheros y registros, en contraposición a bytes o palabras individuales. El disco se utiliza también para proporcionar una extensión de la memoria principal conocida como memoria virtual, que se estudiará en el Capítulo 8.

De hecho, se pueden añadir niveles adicionales a la jerarquía por software. Por ejemplo, se puede utilizar una parte de la memoria principal como una zona de almacenamiento intermedio para guardar temporalmente datos que van a ser leídos del disco. Esta técnica, denominada a veces cache de disco (que se estudia en detalle en el Capítulo 11), mejora el rendimiento de dos maneras:

- Las escrituras en el disco pueden agruparse. En vez de muchas transferencias de datos de pequeño tamaño, se producen unas pocas transferencias de gran tamaño. Esto mejora el rendimiento del disco y minimiza el grado de implicación del procesador.
- Un programa puede acceder a algunos datos destinados a ser escritos antes del siguiente volcado al disco. En ese caso, los datos se recuperan rápidamente de la cache software en vez de lentamente como ocurre cuando se accede al disco.

El Apéndice 1A estudia las implicaciones en el rendimiento de las estructuras de memoria de múltiples niveles.

1.6. MEMORIA CACHE

Aunque la memoria cache es invisible para el sistema operativo, interactúa con otros elementos del hardware de gestión de memoria. Además, muchos de los principios usados en los esquemas de memoria virtual (estudiados en el Capítulo 8) son aplicables también a la memoria cache.

MOTIVACIÓN

En todos los ciclos de instrucción, el procesador accede a memoria al menos una vez, para leer la instrucción y, con frecuencia, una o más veces adicionales, para leer y/o almacenar los resultados. La velocidad a la que el procesador puede ejecutar instrucciones está claramente limitada por el tiempo de ciclo de memoria (el tiempo que se tarda en leer o escribir una palabra de la memoria). Esta limitación ha sido de hecho un problema significativo debido a la persistente discrepancia entre la velocidad del procesador y la de la memoria principal; a lo largo de los años, la velocidad del procesador se ha incrementado constantemente de forma más rápida que la velocidad de acceso a la memoria. El diseñador se encuentra con un compromiso entre velocidad, coste y tamaño. Idealmente, se debería construir la memoria principal con la misma tecnología que la de los registros del procesador, consiguiendo tiempos de ciclo de memoria comparables a los tiempos de ciclo del procesador. Esta estrategia siempre ha resultado demasiado costosa. La solución consiste en aprovecharse del principio de la proximidad utilizando una memoria pequeña y rápida entre el procesador y la memoria principal, denominada cache.

FUNDAMENTOS DE LA CACHE

El propósito de la memoria cache es proporcionar un tiempo de acceso a memoria próximo al de las memorias más rápidas disponibles y, al mismo tiempo, ofrecer un tamaño de memoria grande que tenga el precio de los tipos de memorias de semiconductores menos costosas. El concepto se muestra en la Figura 1.16. Hay una memoria principal relativamente grande y lenta junto con una memoria cache más pequeña y rápida. La cache contiene una copia de una parte de la memoria principal. Cuando el procesador intenta leer un byte de la memoria, se hace una comprobación para determinar si el byte está en la cache. Si es así, se le entrega el byte al procesador. En caso contrario, se lee e introduce dentro de la cache un bloque de memoria principal, que consta de un cierto número fijo de bytes, y, a continuación, se le entrega el byte pedido al procesador. Debido al fenómeno de la proximidad de referencias, cuando se lee e introduce dentro de la cache un bloque de datos para satisfacer una única referencia de memoria, es probable que muchas de las referencias a memoria en el futuro próximo correspondan con otros bytes del bloque.

La Figura 1.17 representa la estructura de un sistema de memoria cache/principal. La memoria principal consta de hasta 2^n palabras direccionables, teniendo cada palabra una única dirección de *n*-bits. A efectos de correspondencia entre niveles, se considera que esta memoria consta de un número de **bloques** de longitud fija de K palabras cada uno. Es decir, hay $M = 2^n/K$ bloques. La cache consiste en C **huecos** (denominados también *líneas*) de K palabras cada uno, tal que el número de huecos es considerablemente menor que el número de bloques de la memoria principal ($C << M$)[6]. Algunos sub-

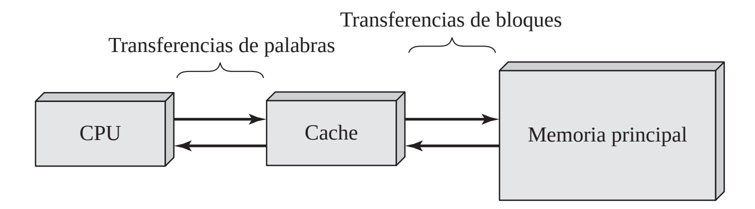


Figura 1.16. Cache y memoria principal.

[6] El símbolo << significa *mucho menor que*. De manera similar, el símbolo >> significa *mucho mayor que*.

conjuntos de los bloques de memoria principal residen en los huecos de la cache. Si se lee una palabra de un bloque de memoria que no está en la cache, se transfiere ese bloque a uno de los huecos de la cache. Dado que hay más bloques que huecos, no se puede dedicar un hueco individual de forma única y permanente a un determinado bloque. Por tanto, cada hueco incluye una etiqueta que identifica qué bloque en concreto se almacena actualmente en él. La etiqueta corresponde normalmente con varios bits de la parte de mayor peso de la dirección y hace referencia a todas las direcciones que comienzan con esa secuencia de bits.

Como un ejemplo sencillo, supóngase una dirección de seis bits y una etiqueta de 2 bits. La etiqueta 01 se refiere al bloque de posiciones con las siguientes direcciones: 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110 y 011111.

La Figura 1.18 muestra la operación de lectura. El procesador genera la dirección DL de la palabra que pretende leer. Si la cache contiene la palabra, se la entrega al procesador. En caso contrario, se carga en la cache el bloque que contiene esa palabra, proporcionando al procesador dicha palabra.

DISEÑO DE LA CACHE

Un estudio detallado del diseño de la cache queda fuera del alcance de este libro. A continuación, se resumen brevemente los elementos fundamentales. Se comprobará más adelante que hay que afrontar

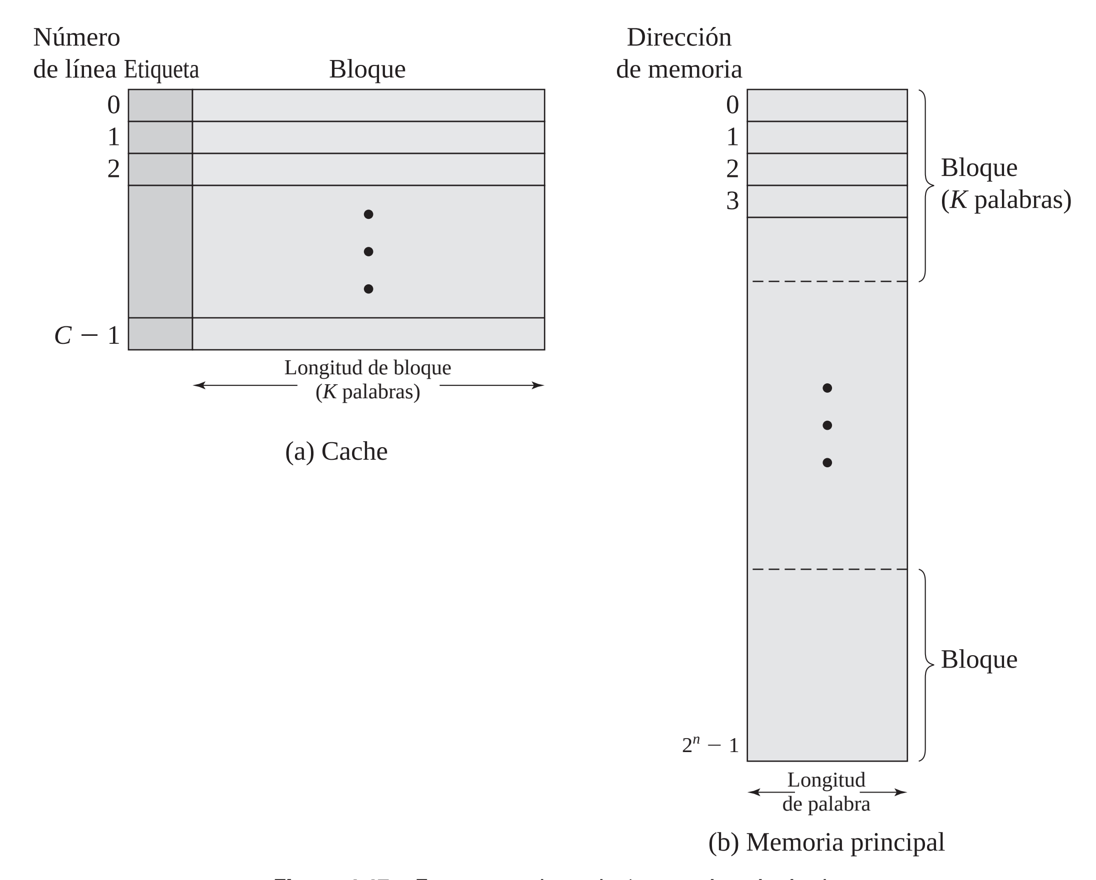


Figura 1.17. Estructura de cache/memoria principal.

aspectos de diseño similares al tratar el diseño de la memoria virtual y de la cache de disco. Se dividen en las siguientes categorías:

- Tamaño de la cache.
- Tamaño del bloque.
- Función de correspondencia.
- Algoritmo de remplazo.
- Política de escritura.

Se ha tratado ya el tema del **tamaño de la cache**, llegándose a la conclusión de que una cache de un tamaño razonablemente pequeño puede tener un impacto significativo en el rendimiento. Otro aspecto relacionado con la capacidad de la cache es el **tamaño del bloque**: la unidad de datos que se intercambia entre la cache y la memoria principal. Según el tamaño del bloque se incrementa desde muy pequeño a tamaños mayores, al principio la tasa de aciertos aumentará debido al principio de la proximidad: la alta probabilidad de que accedan en el futuro inmediato a los datos que están en la proximidad de una palabra a la que se ha hecho referencia. Según se incrementa el tamaño de bloque, se llevan a la cache más datos útiles. Sin embargo, la tasa de aciertos comenzará a decrecer cuando el tamaño del bloque sigua creciendo, ya que la probabilidad de volver a usar los datos recientemente

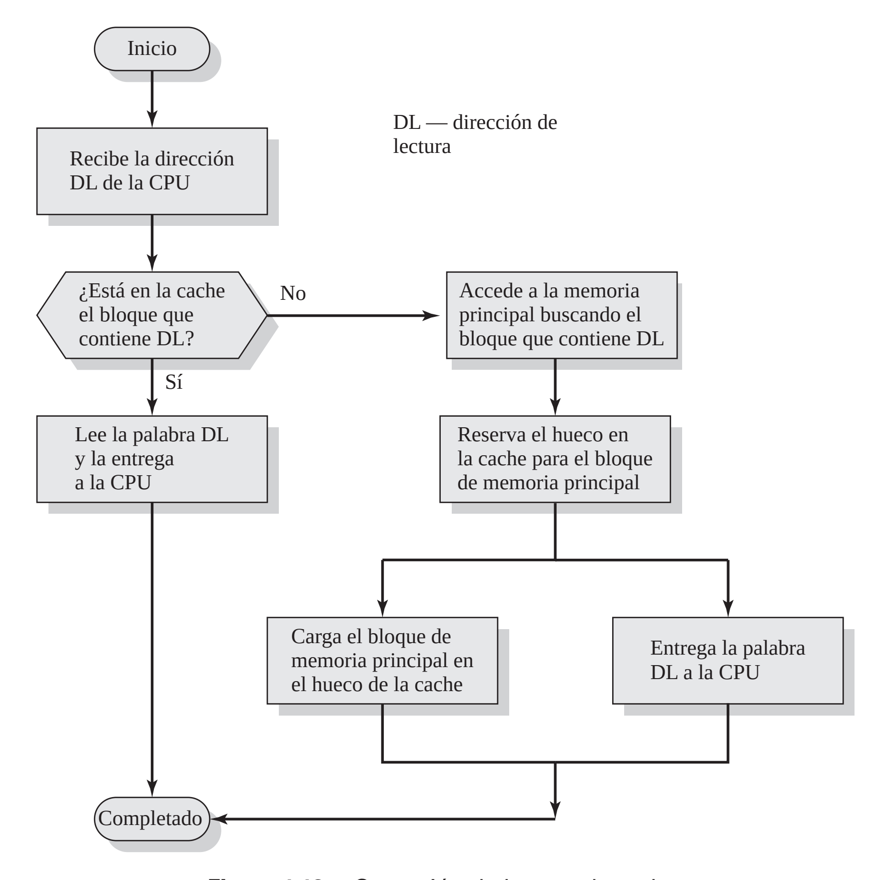


Figura 1.18. Operación de lectura de cache.

leídos se hace menor que la de utilizar nuevamente los datos que se van a expulsar de la cache para dejar sitio al nuevo bloque.

Cuando se lee e incluye un nuevo bloque de datos en la cache, la **función de correspondencia** determina qué posición de la cache ocupará el bloque. Existen dos restricciones que afectan al diseño de la función de correspondencia.

En primer lugar, cuando se introduce un bloque en la cache, se puede tener que remplazar otro. Sería deseable hacer esto de manera que se minimizara la probabilidad de que se remplazase un bloque que se necesitara en el futuro inmediato. Cuanto más flexible es la función de correspondencia, mayor grado de libertad a la hora de diseñar un algoritmo de remplazo que maximice la tasa de aciertos. En segundo lugar, cuanto más flexible es la función de correspondencia, más compleja es la circuitería requerida para buscar en la cache y determinar si un bloque dado está allí.

El **algoritmo de remplazo** selecciona, dentro de las restricciones de la función de correspondencia, qué bloque remplazar cuando un nuevo bloque va a cargarse en la cache y ésta tiene todos los huecos llenos con otros bloques. Sería deseable remplazar el bloque que menos probablemente se va a necesitar de nuevo en el futuro inmediato. Aunque es imposible identificar tal bloque, una estrategia razonablemente eficiente es remplazar el bloque que ha estado en la cache durante más tiempo sin haberse producido ninguna referencia a él. Esta política se denomina el algoritmo del *menos recientemente usado* (*Least Recently Used*, LRU). Se necesitan mecanismos hardware para identificar el bloque menos recientemente usado.

Si se altera el contenido de un bloque en la cache, es necesario volverlo a escribir en la memoria principal antes de remplazarlo. La **política de escritura** dicta cuando tiene lugar la operación de escritura en memoria. Una alternativa es que la escritura se produzca cada vez que se actualiza el bloque. Otra opción es que la escritura se realice sólo cuando se remplaza el bloque. La última estrategia minimiza las operaciones de escritura en memoria pero deja la memoria principal temporalmente en un estado obsoleto. Esto puede interferir con el modo de operación de un multiprocesador y con el acceso directo a memoria realizado por los módulos hardware de E/S.

1.7. TÉCNICAS DE COMUNICACIÓN DE E/S

Hay tres técnicas para llevar a cabo las operaciones de E/S:

- E/S programada.
- E/S dirigida de interrupciones.
- Acceso directo a memoria (*Direct Memory Access*, DMA).

E/S PROGRAMADA

Cuando el procesador ejecuta un programa y encuentra una instrucción relacionada con la E/S, ejecuta esa instrucción generando un mandato al módulo de E/S apropiado. En el caso de la E/S programada, el módulo de E/S realiza la acción solicitada y fija los bits correspondientes en el registro de estado de E/S, pero no realiza ninguna acción para avisar al procesador. En concreto, no interrumpe al procesador. Por tanto, después de que se invoca la instrucción de E/S, el procesador debe tomar un papel activo para determinar cuándo se completa la instrucción de E/S. Por este motivo, el procesador comprueba periódicamente el estado del módulo de E/S hasta que encuentra que se ha completado la operación.

Con esta técnica, el procesador es responsable de extraer los datos de la memoria principal en una operación de salida y de almacenarlos en ella en una operación de entrada. El sotfware de E/S se escribe de manera que el procesador ejecuta instrucciones que le dan control directo de la operación de E/S, incluyendo comprobar el estado del dispositivo, enviar un mandato de lectura o de escritura, y transferir los datos. Por tanto, el juego de instrucciones incluye instrucciones de E/S de las siguientes categorías:

- **Control.** Utilizadas para activar un dispositivo externo y especificarle qué debe hacer. Por ejemplo, se le puede indicar a una unidad de cinta magnética que se rebobine o avance un registro.
- **Estado.** Utilizadas para comprobar diversas condiciones de estado asociadas a un módulo de E/S y sus periféricos.
- **Transferencia.** Utilizadas para leer y/o escribir datos entre los registros del procesador y los dispositivos externos.

La Figura 1.19a proporciona un ejemplo del uso de E/S programada para leer un bloque de datos de un dispositivo externo (p. ej. un registro de cinta) y almacenarlo en memoria. Los datos se leen palabra a palabra (por ejemplo, 16 bits). Por cada palabra que se lee, el procesador debe permanecer en un bucle de comprobación del estado hasta que determina que la palabra está disponible en el registro de datos del módulo de E/S. Este diagrama de flujo subraya las desventajas principales de esta técnica: es un proceso que consume un tiempo apreciable que mantiene al procesador ocupado innecesariamente.

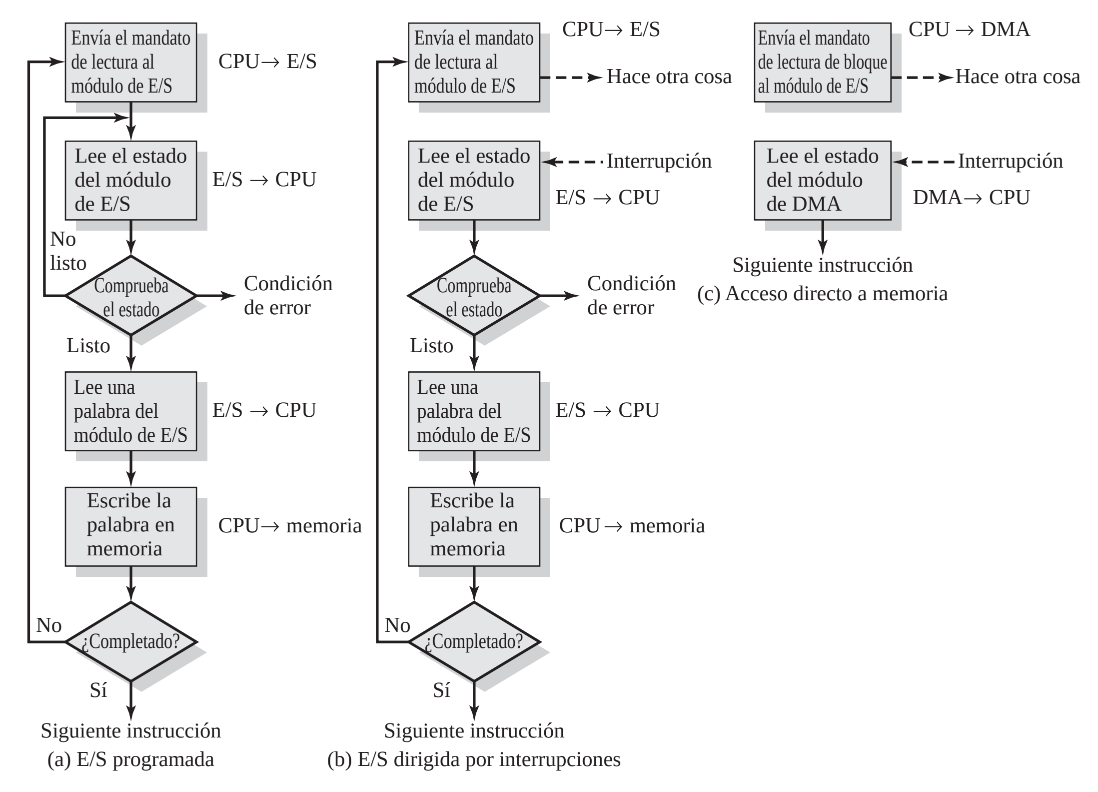


Figura 1.19. Tres técnicas para leer un bloque de datos.

E/S DIRIGIDA POR INTERRUPCIONES

El problema de la E/S programada es que el procesador tiene que esperar mucho tiempo hasta que el módulo de E/S correspondiente esté listo para la recepción o la transmisión de más datos. El procesador, mientras está esperando, debe comprobar repetidamente el estado del módulo de E/S. Como resultado, el nivel de rendimiento de todo el sistema se degrada gravemente.

Una alternativa es que el procesador genere un mandato de E/S para un módulo y, acto seguido, continúe realizando algún otro trabajo útil. El módulo de E/S interrumpirá más tarde al procesador para solicitar su servicio cuando esté listo para intercambiar datos con el mismo. El procesador ejecutará la transferencia de datos, como antes, y después reanudará el procesamiento previo.

Considere cómo funciona esta alternativa, primero desde el punto de vista del módulo de E/S. Para una operación de entrada, el módulo de E/S recibe un mandato de LECTURA del procesador. El módulo de E/S pasa entonces a leer los datos de un periférico asociado. Una vez que los datos están en el registro de datos del módulo, el módulo genera una interrupción al procesador a través de una línea de control. El módulo entonces espera hasta que el procesador pida sus datos. Cuando se hace la petición, el módulo sitúa sus datos en el bus de datos y ya está listo para otra operación de E/S.

Desde el punto de vista del procesador, las acciones correspondientes a una operación de lectura son las que se describen a continuación. El procesador genera un mandato de LECTURA. Salva el contexto (por ejemplo, el contador de programa y los registros del procesador) del programa actual y lo abandona, pasando a hacer otra cosa (por ejemplo, el procesador puede estar trabajando en varios programas diferentes a la vez). Al final de cada ciclo de instrucción, el procesador comprueba si hay interrupciones (Figura 1.7). Cuando se produce la interrupción del módulo de E/S, el procesador salva el contexto del programa que se está ejecutando actualmente y comienza a ejecutar un programa de manejo de interrupción que procesa la interrupción. En este caso, el procesador lee la palabra de datos del módulo de E/S y la almacena en memoria. A continuación, restaura el contexto del programa que había realizado el mandato de E/S (o de algún otro programa) y reanuda su ejecución.

La Figura 1.19b muestra el uso de la E/S dirigida por interrupciones para leer un bloque de datos. La E/S dirigida por interrupciones es más eficiente que la E/S programada ya que elimina la espera innecesaria. Sin embargo, la E/S dirigida por interrupciones todavía consume mucho tiempo de procesador, puesto que cada palabra de datos que va desde la memoria al módulo de E/S o desde el módulo de E/S hasta la memoria debe pasar a través del procesador.

Casi invariablemente, habrá múltiples módulos de E/S en un computador, por lo que se necesitan mecanismos para permitir que el procesador determine qué dispositivo causó la interrupción y para decidir, en caso de múltiples interrupciones, cuál debe manejar primero. En algunos sistemas, hay múltiples líneas de interrupción, de manera que cada módulo de E/S usa una línea diferente. Cada línea tendrá una prioridad diferente. Alternativamente, puede haber una única línea de interrupción, pero se utilizan líneas adicionales para guardar la dirección de un dispositivo. De nuevo, se le asignan diferentes prioridades a los distintos dispositivos.

ACCESO DIRECTO A MEMORIA

La E/S dirigida por interrupciones, aunque más eficiente que la E/S programada simple, todavía requiere la intervención activa del procesador para transferir datos entre la memoria y un módulo de E/S, ya que cualquier transferencia de datos debe atravesar un camino a través del procesador. Por tanto, ambas formas de E/S sufren dos inconvenientes inherentes:

1. La tasa de transferencia de E/S está limitada por la velocidad con la que el procesador puede comprobar el estado de un dispositivo y ofrecerle servicio.
2. El procesador está involucrado en la gestión de una transferencia de E/S; se deben ejecutar varias instrucciones por cada transferencia de E/S.

Cuando se van a transferir grandes volúmenes de datos, se requiere una técnica más eficiente: el acceso directo a memoria (*Direct Memory Access*, DMA). La función de DMA puede llevarla a cabo un módulo separado conectado en el bus del sistema o puede estar incluida en un módulo de E/S. En cualquier caso, la técnica funciona como se describe a continuación. Cuando el procesador desea leer o escribir un bloque de datos, genera un mandato al módulo de DMA, enviándole la siguiente información:

- Si se trata de una lectura o de una escritura.
- La dirección del dispositivo de E/S involucrado.
- La posición inicial de memoria en la que se desea leer los datos o donde se quieren escribir.
- El número de palabras que se pretende leer o escribir.

A continuación, el procesador continúa con otro trabajo. Ha delegado esta operación de E/S al módulo de DMA, que se ocupará de la misma. El módulo de DMA transferirá el bloque completo de datos, palabra a palabra, hacia la memoria o desde ella sin pasar a través del procesador. Por tanto, el procesador solamente está involucrado al principio y al final de la transferencia (Figura 1.19c).

El módulo de DMA necesita tomar el control del bus para transferir datos hacia la memoria o desde ella. Debido a esta competencia en el uso del bus, puede haber veces en las que el procesador necesita el bus y debe esperar al módulo de DMA. Nótese que esto no es una interrupción; el procesador no salva un contexto y pasa a hacer otra cosa. En su lugar, el procesador se detiene durante un ciclo de bus (el tiempo que se tarda en transferir una palabra a través del bus). El efecto global es causar que el procesador ejecute más lentamente durante una transferencia de DMA en el caso de que el procesador requiera acceso al bus. Sin embargo, para una transferencia de E/S de múltiples palabras, el DMA es mucho más eficiente que la E/S dirigida por interrupciones o la programada.

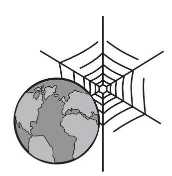
1.8. LECTURAS Y SITIOS WEB RECOMENDADOS

[STAL03] cubre en detalle los temas de este capítulo. Además, hay muchos otros libros sobre arquitectura y organización de computadores. Entre los textos más notables están los siguientes: [PATT98] es un estudio general; [HENN02], de los mismos autores, es un libro más avanzado que enfatiza sobre aspectos cuantitativos de diseño.

HENN02 Hennessy, J., y Patterson, D. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2002.

PATT98 Patterson, D., y Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, CA: Morgan Kaufmann, 1998.

STAL03 Stallings, W. *Computer Organization and Architecture, 6th ed.* Upper Saddle River, NJ: Prentice Hall, 2003.



SITIOS WEB RECOMENDADOS

- **WWW Computer Architecture Home Page.** Un índice amplio de información relevante para los investigadores en arquitectura de computadores, incluyendo grupos y proyectos de arquitectura, organizaciones técnicas, bibliografía, empleo, e información comercial.
- **CPU Info Center.** Información sobre procesadores específicos, incluyendo artículos técnicos, información de productos y los últimos anuncios.

1.9. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

acceso directo a memoria (DMA)	marco de pila	proximidad temporal
bus del sistema	memoria cache	puntero de pila
ciclo de instrucción	memoria principal	puntero de segmento
código de condición	memoria secundaria	registro
contador de programa	módulo de E/S	registro de datos
E/S dirigida por interrupciones	multiprogramación	registro de dirección
E/S programada	pila	registro índice
entrada/salida (E/S)	procedimiento reentrante	registro de instrucción
hueco de cache	procesador	tasa de aciertos
Instrucción	proximidad	unidad central de proceso (CPU)
Interrupción	proximidad espacial	

CUESTIONES DE REPASO

1.1. Enumere y defina brevemente los cuatro elementos principales de un computador.

1.2. Defina las dos categorías principales de los registros del procesador.

1.3. En términos generales, ¿cuáles son las cuatro acciones distintas que puede especificar una instrucción de máquina?

1.4. ¿Qué es una interrupción?

1.5. ¿Cómo se tratan múltiples interrupciones?

1.6. ¿Qué características distinguen a los diversos elementos de una jerarquía de memoria?

1.7. ¿Qué es una memoria cache?

1.8. Enumere y defina brevemente las tres técnicas para las operaciones de E/S.

1.9. ¿Cuál es la diferencia entre la proximidad espacial y la temporal?

1.10. En general, ¿cuáles son las estrategias para aprovechar la proximidad espacial y la temporal?

PROBLEMAS

1.1. Suponga que la máquina hipotética de la Figura 1.3 tiene también dos instrucciones de E/S:

0011 = Carga el AC con un valor leído de un dispositivo de E/S
0111 = Almacena el AC en un dispositivo de E/S

En estos casos, la dirección de 12 bits identifica un determinado dispositivo externo. Muestre la ejecución del programa (utilizando el formato de la Figura 1.4) correspondiente al siguiente fragmento:

1. Carga el AC con un valor leído del dispositivo 5.
2. Suma al AC el contenido de la posición de memoria 940.
3. Almacena el AC en el dispositivo 6.
4. Asuma que el siguiente valor leído del dispositivo 5 es 3 y que la posición 940 contiene el valor 2.

1.2. La ejecución del programa de la Figura 1.4 se describe en el texto utilizando seis pasos. Extienda esta descripción para mostrar el uso del RDIM y del RDAM.

1.3. Considere un hipotético microprocesador de 32 bits que tiene instrucciones de 32 bits compuestas de dos campos: el primer byte contiene el código de operación y el resto un operando inmediato o la dirección de un operando.

a) ¿Cuál es la máxima capacidad de memoria directamente direccionable (en bytes)?
b) Estudie el impacto en la velocidad del sistema dependiendo de si el bus del microprocesador tiene:
   1. un bus de direcciones local de 32 bits y un bus de datos local de 16 bits o
   2. un bus de direcciones local de 16 bits y un bus de datos local de 16 bits.
c) ¿Cuántos bits se necesitan para el contador del programa y para el registro de instrucciones?

1.4. Considere un microprocesador hipotético que genera una dirección de 16 bits (por ejemplo, asuma que el contador de programa y los registros de dirección tienen un ancho de 16 bits) y que tiene un bus de datos de 16 bits.

a) ¿Cuál es el máximo espacio de direcciones de memoria al que el procesador puede acceder directamente si se conecta a una «memoria de 16 bits»?
b) ¿Cuál es el máximo espacio de direcciones de memoria al que el procesador puede acceder directamente si se conecta a una «memoria de 8 bits»?
c) ¿Qué características arquitectónicas permitirán a este microprocesador acceder a un «espacio de E/S» separado?
d) Si una instrucción de entrada/salida puede especificar un número de puerto de E/S de 8 bits, ¿cuántos puertos de E/S de 8 bits puede manejar el microprocesador? ¿Y cuántos de 16 bits? Razone la respuesta.

1.5. Considere un microprocesador de 32 bits, con un bus de datos externo de 16 bits, alimentado por un reloj de entrada de 8 MHz. Asuma que este microprocesador tiene un ciclo de bus cuya duración mínima es igual a cuatro ciclos del reloj de entrada. ¿Cuál es la tasa de transferencia de datos máxima en el bus que este microprocesador puede mantener medida en bytes/s? Para incrementar su rendimiento, ¿sería mejor hacer que su bus de datos externo tenga 32 bits o doblar la frecuencia del reloj externo suministrada al microprocesador?

Detalle cualquier otra suposición que se realice, razonando la misma. *Sugerencia*: determine el número de bytes que se pueden transferir por cada ciclo de bus.

1.6. Considere un computador que contiene un módulo de E/S que controla un sencillo teletipo con impresora y teclado. La CPU contiene los siguientes registros, que están conectados directamente con el bus del sistema:

RENT: Registro de entrada, 8 bits

RSAL: Registro de salida, 8 bits

INE: Indicador de entrada, 1 bit

INS: Indicador de salida, 1 bit

HAI: Habilitación de interrupción, 1 bit

El módulo de E/S controla la entrada de teclado del teletipo y la salida a la impresora. El teletipo es capaz de codificar un símbolo alfanumérico en palabras de 8 bits y descodificar una palabra de 8 bits en un símbolo alfanumérico. El indicador de entrada se activa cuando se introduce una palabra de 8 bits en el registro de entrada del teletipo. El indicador de salida se activa cuando se imprime una palabra.

a) Describa cómo la CPU, utilizando los cuatro primeros registros enumerados en este problema, puede realizar E/S con el teletipo.

b) Describa cómo se puede realizar más eficientemente la función empleando también HAI.

1.7. En prácticamente todos los sistemas que incluyen módulos de DMA, se otorga mayor prioridad a los accesos del módulo de DMA a la memoria principal que a los accesos del procesador. ¿Por qué?

1.8. Un módulo de DMA está transfiriendo caracteres a la memoria principal desde un dispositivo externo transmitiendo a 9600 bits por segundo (bps). El procesador puede leer instrucciones a una velocidad de 1 millón de instrucciones por segundo. ¿En cuánto se ralentizará el procesador debido a la actividad de DMA?

1.9. Un computador consta de una CPU y un dispositivo *D* de E/S conectado a la memoria principal *M* mediante un bus compartido con una anchura de bus de datos de una palabra. La CPU puede ejecutar un máximo de 10^6 instrucciones por segundo. Una instrucción media requiere cinco ciclos de máquina, tres de los cuales utilizan el bus de memoria. Una operación de lectura o escritura en memoria utiliza un ciclo de máquina. Supóngase que la CPU está ejecutando constantemente programas en segundo plano (*background*) que requieren el 95% de su tasa de ejecución de instrucciones pero ninguna instrucción de E/S. Asuma que un ciclo de procesador es igual a un ciclo de bus. Ahora suponga que se tienen que transferir bloques de datos muy grandes entre *M* y *D*.

a) Si se utiliza E/S programada y cada transferencia de E/S de una palabra requiere que la CPU ejecute dos instrucciones, estime la tasa máxima de transferencia de datos posible de E/S, en palabras por segundo, a través de *D*.

b) Estime la misma tasa si se utiliza una transferencia mediante DMA

1.10. Considere el siguiente código:

```
for (i = 0; i < 20; i++)
        for (j = 0; j < 10; j++)
                a[i] = a[i] *j
```

a) Proporcione un ejemplo de proximidad espacial en el código.
b) Proporcione un ejemplo de proximidad temporal en el código.

1.11. Generalice las ecuaciones (1.1) y (1.2) del Apéndice 1A para jerarquías de memoria de *n* niveles.

1.12. Considere un sistema de memoria con los siguientes parámetros:

T_c = 100 ns C_c = 0,01 céntimos/bit

T_m = 1.200 ns C_m = 0,001 céntimos/bit

a) ¿Cuál es el coste de 1MByte de memoria principal?
b) ¿Cuál es el coste de 1Mbyte de memoria principal utilizando tecnología de memoria cache?
c) Si el tiempo de acceso efectivo es un 10% mayor que el tiempo de acceso a la cache, ¿cuál es la tasa de aciertos *A*?

1.13. Un computador tiene una cache, una memoria principal y un disco usado para la memoria virtual. Si la palabra accedida está en la cache, se requieren 20 ns para accederla. Si está en la memoria principal pero no en la cache, se necesitan 60 ns para cargarla en la cache (esto incluye el tiempo para comprobar inicialmente si está en la cache) y, a continuación, la referencia comienza de nuevo. Si la palabra no está en memoria principal, se requieren 12 ms para buscar la palabra del disco, seguido de 60 ns para copiarla de la cache y luego se inicia nuevamente la referencia. La tasa de aciertos de la cache es 0,9 y la de la memoria principal es 0,6. ¿Cuál es el tiempo medio en ns requerido para acceder a una palabra en este sistema?

1.4. Suponga que el procesador utiliza una pila para gestionar las llamadas a procedimiento y los retornos de los mismos. ¿Puede eliminarse el contador de programa utilizando la cima de la pila como contador de programa?

APÉNDICE 1A CARACTERÍSTICAS DE RENDIMIENTO DE LAS MEMORIAS DE DOS NIVELES

En este capítulo, se hace referencia a la cache que actúa como un *buffer* entre la memoria principal y el procesador, creando una memoria interna de dos niveles. Esta arquitectura de dos niveles proporciona un rendimiento mejorado con respecto a una memoria de un nivel equiparable, explotando una propiedad conocida como proximidad, que se analizará en este apéndice.

Tabla 1.2. Características de las memorias de dos niveles.

	Cache de memoria principal	Memoria virtual (Paginación)	Cache de disco
Proporción típica entre tiempos de acceso	5:1	10^6:1	10^6:1
Sistema de gestión de memoria	Implementado por un hardware especial	Combinación de hardware y software de sistema	Software de sistema
Tamaño típico de bloque	4 a 128 bytes	64 a 4096 bytes	64 a 4096 bytes
Acceso del procesador al segundo nivel	Acceso directo	Acceso indirecto	Acceso indirecto

El mecanismo de cache de memoria principal es parte de la arquitectura del computador, implementado en hardware y habitualmente invisible al sistema operativo. Por tanto, en este libro no se trata este mecanismo. Sin embargo, hay otras dos instancias de la técnica de memoria de dos niveles que también explotan la propiedad de la proximidad y que son, al menos parcialmente, implementadas en el sistema operativo: la memoria virtual y la cache de disco (Tabla 1.2). Estos dos temas se analizarán en los Capítulos 8 y 11, respectivamente. En este apéndice, se revisarán algunas características de rendimiento de la memoria de dos niveles que son comunes a las tres técnicas.

PROXIMIDAD

La base de la ganancia en rendimiento de la memoria de dos niveles reside en el principio de la proximidad, comentado en la Sección 1.5. Este principio establece que las referencias a memoria tienden a agruparse. En un largo periodo de tiempo, los grupos que se están usando van cambiando, pero en un periodo corto, el procesador está primordialmente trabajando con grupos fijos de referencias a memoria.

La experiencia real muestra que el principio de la proximidad es válido. Para comprobarlo, considere la siguiente línea de razonamiento:

1. Excepto para las interrupciones de salto y llamada, que constituyen sólo una pequeña parte de todas las instrucciones del programa, la ejecución del programa es secuencial. Por tanto, en la mayoría de los casos, la próxima instrucción que se va a leer es la que sigue inmediatamente a la última instrucción leída.

2. No es frecuente que se produzca una larga secuencia ininterrumpida de llamadas a procedimiento seguida por la correspondiente secuencia de retornos. Lo habitual es que un programa permanezca confinado en una ventana de anidamiento de invocación de procedimientos bastante estrecha. Por tanto, durante un periodo corto de tiempo, las referencias a instrucciones tienden a localizarse en unos pocos procedimientos.

3. La mayoría de las construcciones iterativas consta de un número relativamente pequeño de instrucciones repetidas muchas veces. Mientras se ejecuta una iteración, el cálculo queda confinado, por tanto, en una pequeña parte contigua del programa.

4. En muchos programas, gran parte del cálculo implica procesar estructuras de datos, tales como vectores o secuencias de registros. En muchos casos, las sucesivas referencias a estas estructuras de datos corresponderán con elementos de datos situados próximamente.

Esta línea de razonamiento se ha confirmado en muchos estudios. Con respecto al primer punto, varios estudios han analizado el comportamiento de programas escritos en lenguajes de alto nivel. La Tabla 1.3 incluye los resultados fundamentales, midiendo la aparición de varios tipos de sentencias durante la ejecución, extraídos de los estudios que se detallan a continuación. El más antiguo estudio del comportamiento de un lenguaje de programación, realizado por Knuth [KNUT71], examinaba una colección de programas en FORTRAN usados como ejercicios para estudiantes. Tanenbaum [TANE78] publicó medidas recogidas de unos 300 procedimientos utilizados en programas del sistema operativo y escritos en un lenguaje que da soporte a la programación estructurada (SAL). Patterson y Sequin [PATT82] analizaron un conjunto de medidas obtenidas por compiladores, programas de composición de documentos, de diseño asistido por computador (*Computer-Aided Design*, CAD), de ordenamiento y de comparación de ficheros. Se estudiaron los lenguajes de programación C y Pascal. Huck [HUCK83] analizó cuatro programas seleccionados para representar una mezcla de cálculos científicos de propósito general, incluyendo la transformada rápida de Fourier y la integración de sistemas de ecuaciones diferenciales. Hay una coincidencia general en los resultados de esta mezcla

de lenguajes y aplicaciones en lo que se refiere a que las instrucciones de salto y de llamada representan sólo una fracción de las sentencias ejecutadas durante la vida de un programa. Por tanto, estos estudios confirman la primera afirmación de la lista precedente.

Tabla 1.3. Frecuencia dinámica relativa de las operaciones en lenguajes de alto nivel.

Estudio Lenguaje Tipo de carga	[HUCK83] Pascal Científica	[KNUT71] FORTRAN Estudiantes	[PATT82] Pascal Sistema	 C Sistema	[TANE78] SAL Sistema
Asignación	74	67	45	38	42
Bucle	4	3	5	3	4
Llamada	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Otros	—	7	6	1	6

Con respecto a la segunda afirmación, los estudios presentados en [PATT85] proporcionan una confirmación. Esto se ilustra en la Figura 1.20, que muestra el comportamiento de llamadas y retornos. Cada llamada se representa por una línea descendente que se desplaza hacia la derecha, y cada retorno por una línea ascendente desplazándose a la derecha. En la figura, se define una *ventana* con una profundidad igual a 5. Sólo una secuencia de llamadas y retornos con un movimiento neto de 6 en cualquier dirección causa que la ventana se mueva. Como puede observarse, el programa en ejecución puede permanecer dentro de una ventana estacionaria durante largos periodos de tiempo. Un estudio de los mismos autores sobre programas en C y Pascal mostró que una ventana de profundidad 8 sólo necesitaría desplazarse en menos del 1% de las llamadas o retornos [TAMI83].

El principio de proximidad de referencias continúa siendo validado en los estudios más recientes. Por ejemplo, la Figura 1.21 muestra el resultado de un estudio de los patrones de acceso a las páginas web de un determinado sitio.

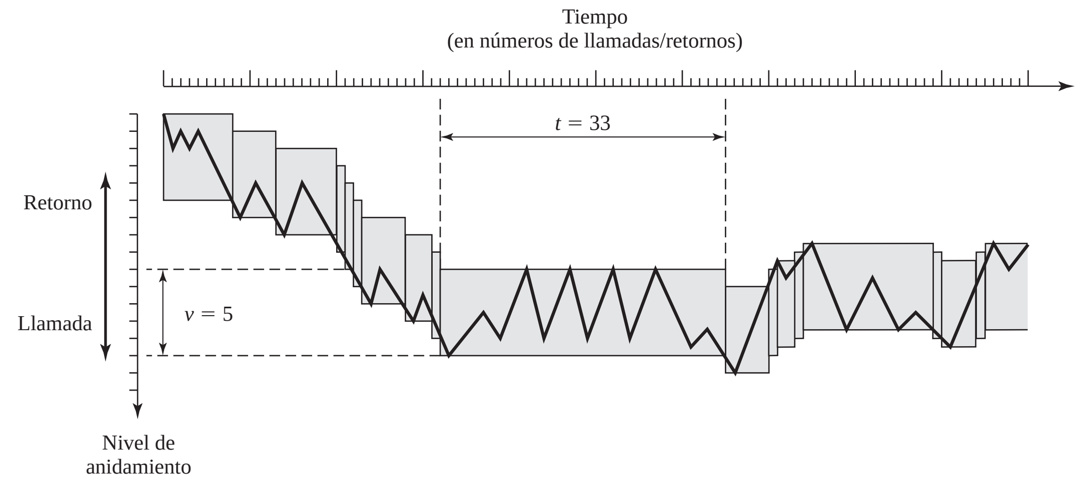


Figura 1.20. Ejemplo de comportamiento de llamadas y retornos de un programa.

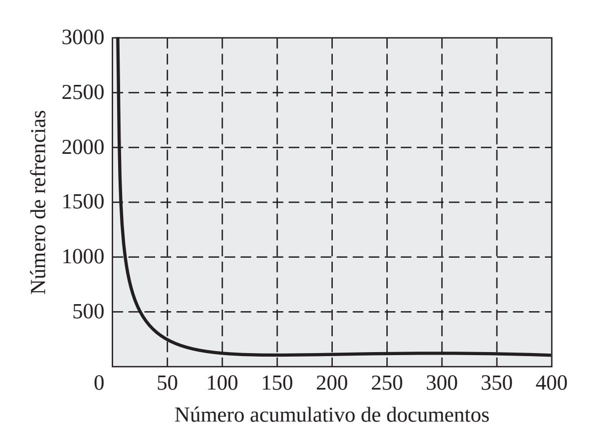


Figura 1.21. Proximidad de referencias para páginas web [BAEN97].

La bibliografía sobre el tema hace una distinción entre la proximidad espacial y la temporal. La **proximidad espacial** se refiere a la tendencia de una ejecución a involucrar posiciones de memoria que están agrupadas. Esto refleja la tendencia de un procesador a acceder secuencialmente a las instrucciones. La proximidad espacial también refleja la tendencia de un programa a acceder de forma secuencial a las posiciones de datos, como cuando se procesa una tabla de datos. La **proximidad temporal** hace referencia a la tendencia de un procesador a acceder a posiciones de memoria que se han utilizado recientemente. Por ejemplo, cuando se ejecuta un bucle, el procesador ejecuta el mismo juego de instrucciones repetidamente.

Tradicionalmente, la proximidad temporal se explota manteniendo en la memoria cache los valores de las instrucciones y los datos usados recientemente aprovechando una jerarquía de cache. La proximidad espacial se explota generalmente utilizando bloques de cache más grandes e incorporando mecanismos de lectura anticipada (se buscan elementos cuyo uso se prevé) en la lógica de control de la cache. Recientemente, ha habido investigaciones considerables para la mejora de estas técnicas con objeto de alcanzar un mayor rendimiento, pero las estrategias básicas siguen siendo las mismas.

MODO DE OPERACIÓN DE LA MEMORIA DE DOS NIVELES

La propiedad de la proximidad se puede explotar para la creación de una memoria de dos niveles. La memoria de nivel superior (M1) es más pequeña, más rápida y más cara (por bit) que la memoria de nivel inferior (M2). M1 se utiliza como un almacenamiento temporal para una parte del contenido de M2, que es más grande. Cuando se realiza una referencia a memoria, se hace un intento de acceder al elemento en M1. Si tiene éxito, se lleva a cabo un acceso rápido. En caso contrario, se copia un bloque de posiciones de memoria de M2 a M1 y, a continuación, el acceso tiene lugar en M1. Gracias a la proximidad, una vez que se trae un bloque a M1, debería haber varios accesos a las posiciones en ese bloque, dando como resultado un servicio global rápido.

Para expresar el tiempo medio de acceso a un elemento, no sólo se deberá considerar las velocidades de los dos niveles de memoria sino también la probabilidad de que una referencia dada puede encontrarse en M1. Se tiene:

$$\begin{aligned} T_S &= A \times T_1 + (1 - A) \times (T_1 + T_2) \\ &= T_1 + (1 - A) \times T_2 \end{aligned} \tag{1.1}$$

donde:

T_S = tiempo medio de acceso (del sistema).

T_1 = tiempo de acceso a M1 (por ejemplo, cache, cache de disco).

T_2 = tiempo de acceso a M2 (por ejemplo, memoria principal, disco).

A = tasa de aciertos (tasa de referencias encontradas en M1).

La Figura 1.15 muestra el tiempo medio de acceso como una función de la tasa de aciertos. Como puede observarse, para un porcentaje alto de aciertos, el tiempo medio de acceso total está mucho más cerca al de M1 que al de M2.

RENDIMIENTO

A continuación, se examinan algunos de los parámetros relevantes para la valoración de un mecanismo de memoria de dos niveles. En primer lugar, se considera el coste:

$$C_S = \frac{C_1 D_1 + C_2 D_2}{D_1 + D_2} \tag{1.2}$$

donde:

C_S = coste medio por bit de la memoria combinada de dos niveles

C_1 = coste medio por bit de la memoria M1 de nivel superior

C_2 = coste medio por bit de la memoria M2 de nivel inferior

D_1 = tamaño de M1

D_2 = tamaño de M2

Sería deseable que $C_S \approx C_2$. Dado que $C_1 >> C_2$, se requiere que $M_1 << M_2$. La Figura 1.22 muestra la relación[7].

Seguidamente, considere el tiempo de acceso. Para que una memoria de dos niveles proporcione una mejora significativa de rendimiento, se necesita tener T_S aproximadamente igual a T_1 ($T_S \approx T_1$). Dado que T_1 es mucho menor que T_2 ($T_1 << T_2$), se necesita una tasa de aciertos próxima a 1.

Por consiguiente, se pretende que M1 sea pequeña para mantener el coste bajo y grande para mejorar la tasa de acierto y, por tanto, el rendimiento. ¿Hay un tamaño de M1 que satisface ambos requisitos hasta un punto razonable? Se puede responder a esta cuestión mediante una serie de preguntas adicionales:

- ¿Qué valor de la tasa de aciertos se necesita para satisfacer el requisito de rendimiento planteado?
- ¿Qué tamaño de M1 asegurará la tasa de aciertos requerida?
- ¿Satisface este tamaño el requisito de coste?

---

[7] Nótese que los dos ejes usan una escala logarítmica. Una revisión básica de las escalas logarítmicas se encuentra en el documento de repaso de matemáticas en el *Computer Science Student Support Site* en **WilliamStallings.com/StudentSupport.html**..

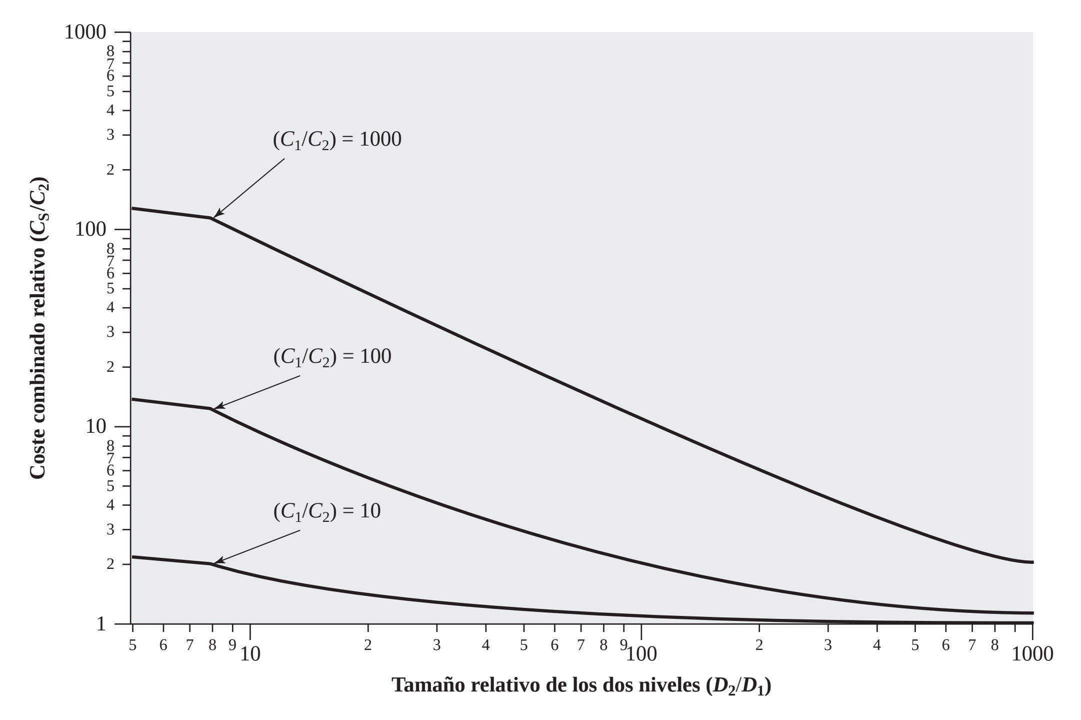


Figura 1.22. Relación del coste medio de la memoria en relación con su tamaño relativo para una memoria de dos niveles.

Para responder a estas preguntas, considere la cantidad T_1/T_S, que se conoce como *eficiencia de acceso*. Es una medida que refleja hasta qué punto el tiempo de acceso medio (T_S) está cerca del tiempo de acceso de M1 (T_1). A partir de la ecuación (1.1):

$$\frac{T_1}{T_S} = \frac{1}{1 + (1 - A)\dfrac{T_2}{T_1}} \qquad (1.3)$$

En la Figura 1.23 se dibuja T_1/T_S como una función de la tasa de aciertos A, con la cantidad T_2/T_1 como parámetro. Parece necesitarse una tasa de aciertos en el intervalo de 0,8 a 0,9 para satisfacer el requisito de rendimiento.

En este momento se puede expresar más exactamente la pregunta sobre el tamaño relativo de la memoria. ¿Es razonable una tasa de aciertos mayor o igual a 0,8 para $D_1 << D_2$? Esto dependerá de varios factores, incluyendo las características del software que se está ejecutando y los detalles de diseño de la memoria de dos niveles. El determinante principal es, evidentemente, el grado de proximidad. La Figura 1.24 sugiere el efecto de la proximidad en la tasa de aciertos. Claramente, si M1 tiene el mismo tamaño que M2, la tasa de aciertos será igual a 1,0: todos los elementos en M2 también estarán siempre almacenados en M1. A continuación, supóngase que no hay proximidad, es decir, las referencias son completamente aleatorias. En ese caso, la tasa de aciertos será una función estrictamente lineal del tamaño relativo de la memoria. Por ejemplo, si M1 tiene la mitad de tamaño de M2, en cualquier momento la mitad de los elementos de M2 están también en M1 y la tasa de aciertos será de 0,5. Sin embargo, en la práctica, hay algún grado de proximidad en las referencias. En la figura se indican los efectos de una proximidad moderada y de una acusada.

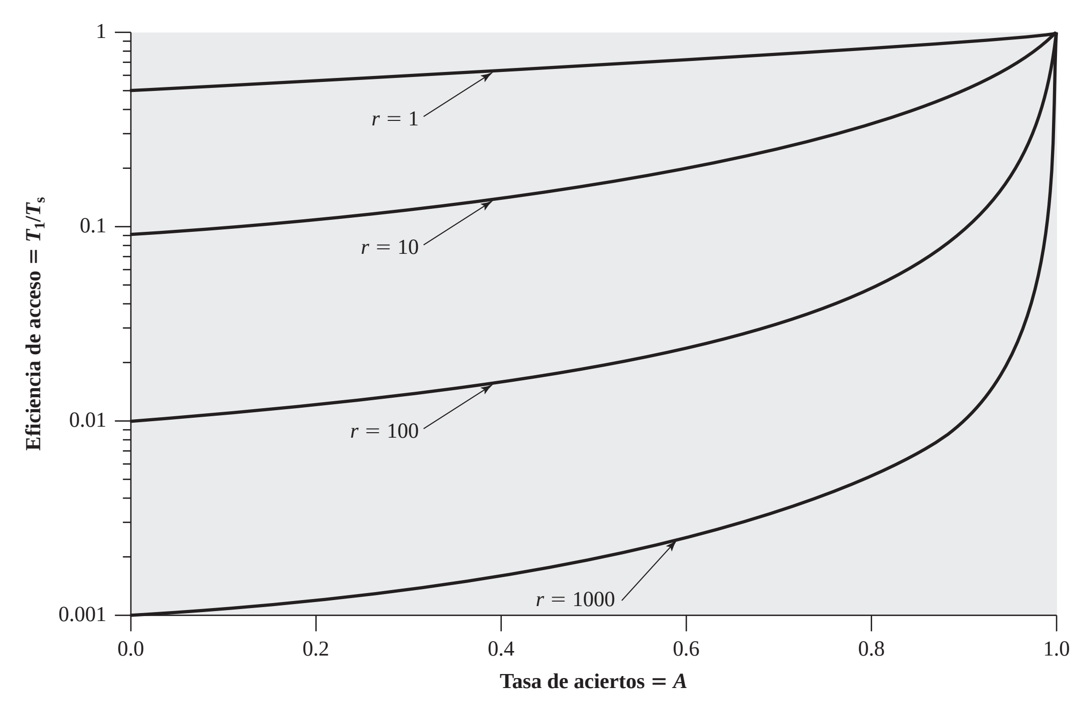


Figura 1.23. Eficiencia de acceso en función de la tasa de aciertos ($r = T_2/T_1$).

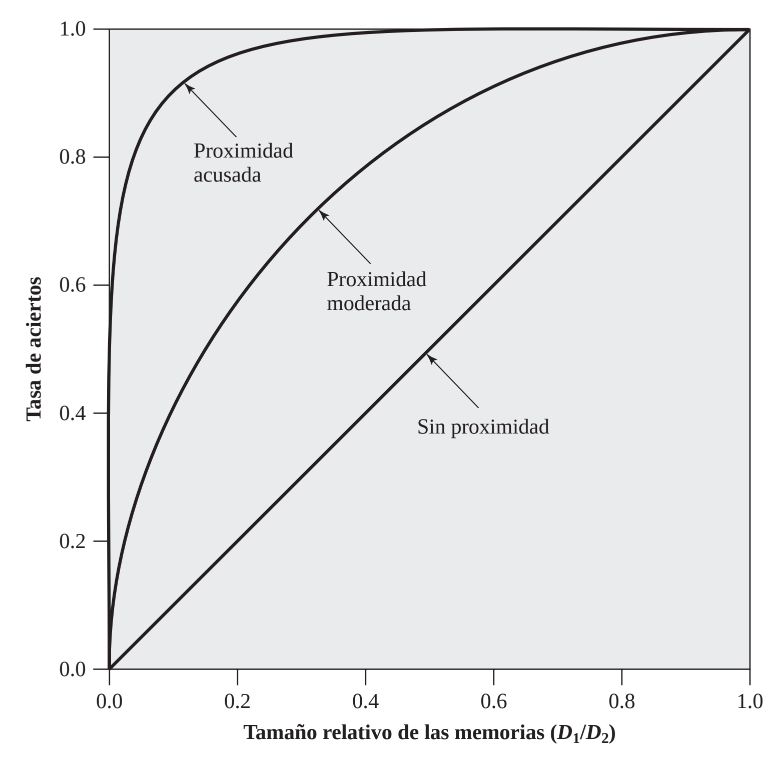


Figura 1.24. Tasa de aciertos en función del tamaño de la memoria.

En consecuencia, si hay una proximidad acusada, es posible lograr valores elevados en la tasa de aciertos incluso con un tamaño de memoria de nivel superior relativamente pequeño. Por ejemplo, numerosos estudios muestran que tamaños de cache bastante pequeños producirán una tasa de aciertos por encima del 0,75, *con independencia del tamaño de la memoria principal* (por ejemplo, [AGAR98], [PRZY88], [STRE83] y [SMIT82]). Una cache con un tamaño en el intervalo entre 1K y 128K palabras es generalmente adecuada, mientras que actualmente la memoria principal se presenta normalmente en un intervalo que se extiende entre cientos de megabytes hasta más de un gigabyte. Cuando se considera la memoria virtual y la cache de disco, se pueden citar otros estudios que confirman el mismo fenómeno, es decir, que una memoria M1 relativamente pequeña produce un valor elevado en la tasa de aciertos gracias a la proximidad.

Esto conduce a la última pregunta listada anteriormente: ¿satisface el tamaño relativo de las dos memorias el requisito de coste? La respuesta es claramente sí. Si se necesita sólo una memoria de nivel superior relativamente pequeña para alcanzar un buen rendimiento, el coste medio por bit de los dos niveles de memoria se aproximará al de la memoria de nivel inferior.

APÉNDICE 1B CONTROL DE PROCEDIMIENTOS

Una técnica habitual para controlar la ejecución de llamadas a procedimiento y los retornos de los mismos es utilizar una pila. Este apéndice resume las propiedades básicas de las pilas y revisa su uso para el control de procedimientos.

IMPLEMENTACIÓN DE LA PILA

Una pila es un conjunto ordenado de elementos, tal que en cada momento solamente se puede acceder a uno de ellos (el más recientemente añadido). El punto de acceso se denomina *cima* de la pila El número de elementos de la pila, o longitud de la pila, es variable. Por esta razón, se conoce también a una pila como una *lista de apilamiento* o *lista donde el último que entra es el primero que sale* (*Last-In First-Out*, LIFO).

La implementación de una pila requiere que haya un conjunto de posiciones dedicado a almacenar los elementos de la pila. En la Figura 1.25 se muestra una técnica habitual. Se reserva en memoria principal (o memoria virtual) un bloque contiguo de posiciones. La mayoría de las veces el bloque está parcialmente lleno con elementos de la pila y el resto está disponible para el crecimiento de la pila. Se necesitan tres direcciones para un funcionamiento adecuado, que habitualmente se almacenan en registros del procesador:

- **Puntero de pila.** Contiene la dirección de la cima de la pila. Si se añade un elemento (APILA) o se elimina (EXTRAE), el puntero se decrementa o se incrementa para contener la dirección de la nueva cima de la pila.
- **Base de la pila.** Contiene la dirección de la posición inferior en el bloque reservado. Se trata de la primera posición que se utiliza cuando se añade un elemento a una pila vacía. Si se hace un intento de extraer un elemento cuando la pila está vacía, se informa del error.
- **Límite de la pila.** Contiene la dirección del otro extremo, o cima, del bloque reservado. Si se hace un intento para apilar un elemento cuando la pila está llena, se indica el error.

Tradicionalmente, y en la mayoría de las máquinas actuales, la base de la pila está en el extremo con la dirección más alta del bloque de pila reservado, y el límite está en el extremo con la dirección más baja. Por tanto, la pila crece desde las direcciones más altas a las más bajas.

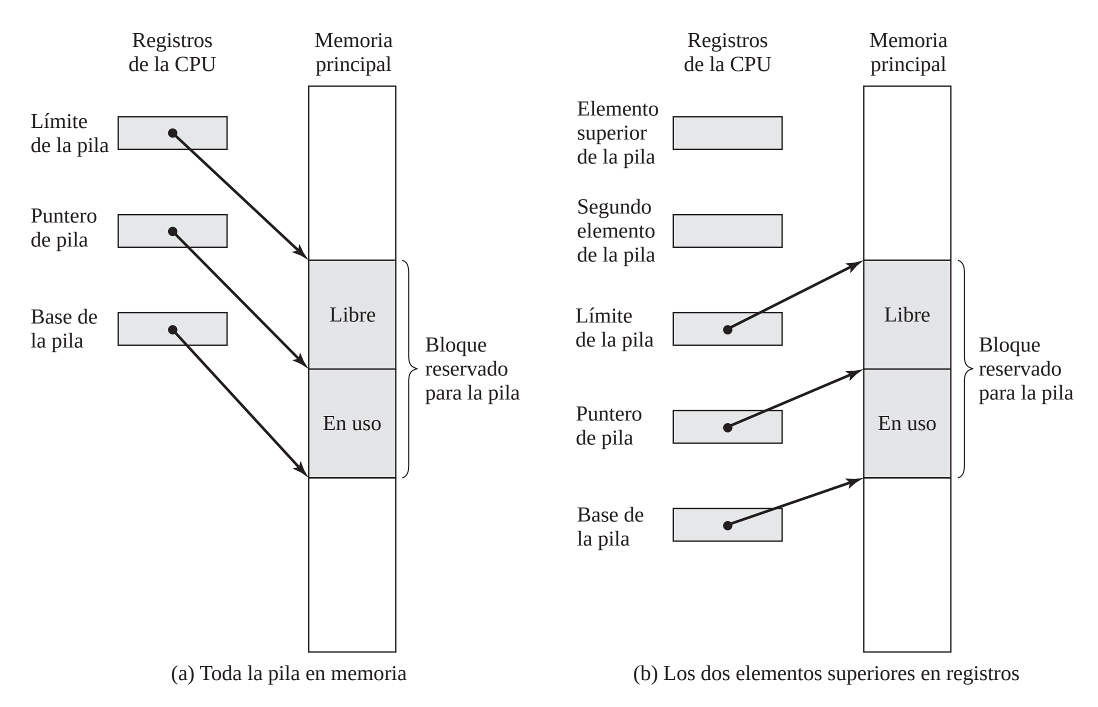


Figura 1.25. Organización habitual de la pila.

LLAMADAS Y RETORNOS DE PROCEDIMIENTOS

Una técnica habitual para gestionar las llamadas y los retornos de los procedimientos es utilizar una pila. Cuando el procesador ejecuta una llamada, se almacena (apila) la dirección de retorno en la pila. Cuando se ejecuta un retorno, se utiliza la dirección de la cima de la pila y se elimina (extrae) esa dirección de la pila. La Figura 1.27 muestra el uso de la pila para los procedimientos anidados presentados en la Figura 1.26.

Es también necesario con frecuencia pasar parámetros en una llamada a procedimiento. Estos se podrían pasar en los registros. Otra posibilidad es almacenar los parámetros en la memoria justo después de las instrucciones de llamada. En este caso, el retorno debe estar en la posición siguiente a los parámetros. Ambas técnicas tienen sus inconvenientes. Si se utilizan los registros, el programa llamado y el que realiza la llamada deben escribirse de manera que se asegure que los registros se utilizan apropiadamente. El almacenamiento de parámetros en memoria hace difícil intercambiar un número variable de parámetros.

Una estrategia más flexible para el paso de parámetros es la pila. Cuando el procesador ejecuta una llamada, no sólo apila la dirección de retorno, sino también los parámetros que se desean pasar al procedimiento llamado. El procedimiento invocado puede acceder a los parámetros en la pila. Al retornar, los parámetros de retorno se pueden almacenar también en la pila, *debajo* de la dirección de retorno. El conjunto completo de parámetros, incluyendo la dirección de retorno, que se almacena en una invocación de procedimiento se denomina **marco de pila**.

En la Figura 1.28 se muestra un ejemplo. El ejemplo se refiere a un procedimiento P en el que se declaran las variables locales *x*1 y *x*2, y el procedimiento Q, al cual puede llamar P y en el que se declaran las variables *y*1 y *y*2. El primer elemento almacenado en cada marco de pila es un puntero al principio del marco previo. Esta técnica se necesita si el número o la longitud de los parámetros que

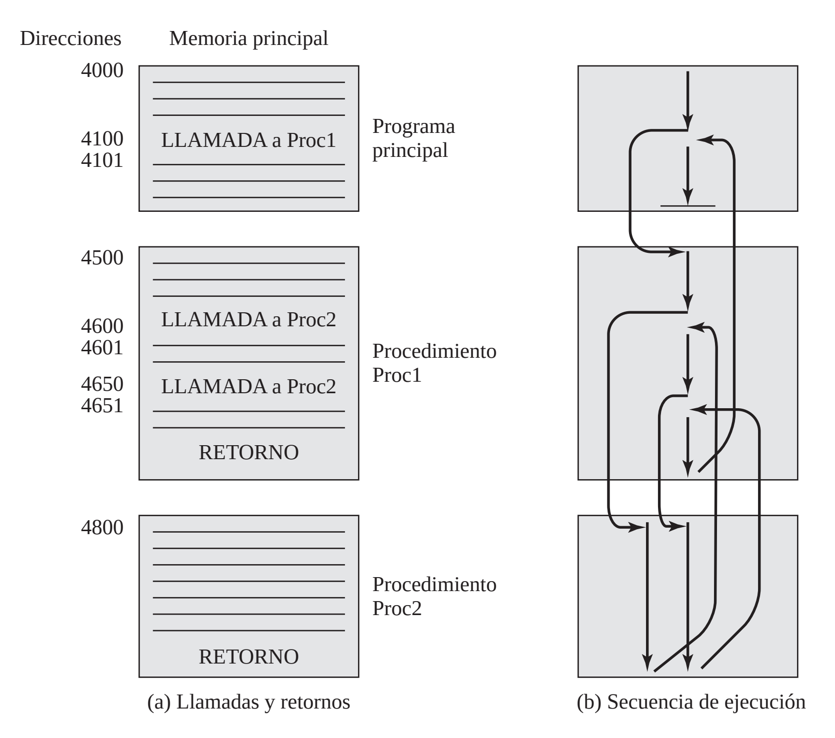


Figura 1.26. Procedimientos anidados.

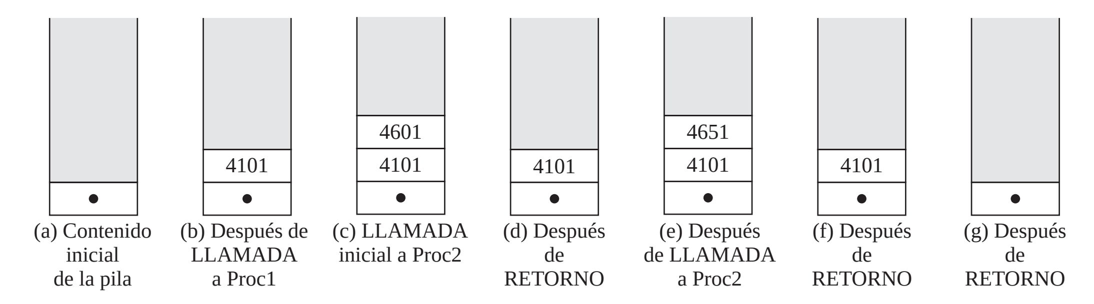


Figura 1.27. Uso de la pila para implementar los procedimientos anidados de la Figura 1.26.

se van a apilar es variable. A continuación, se almacena el punto de retorno del procedimiento que corresponde a este marco de pila. Finalmente, se reserva espacio en la cima del marco de pila para las variables locales. Estas variables locales se pueden utilizar para el paso de parámetros. Por ejemplo, supóngase que cuando P llama a Q le pasa un valor como parámetro. Este valor se podría almacenar en la variable *y*1. Por tanto, en un lenguaje de alto nivel, habría una instrucción en la rutina P similar a la siguiente:

LLAMADA Q(*y*1)

Cuando se ejecuta esta llamada, se crea un nuevo marco de pila para Q (Figura 1.28b), que incluye un puntero al marco de pila para P, la dirección de retorno de P, y dos variables locales para Q, una

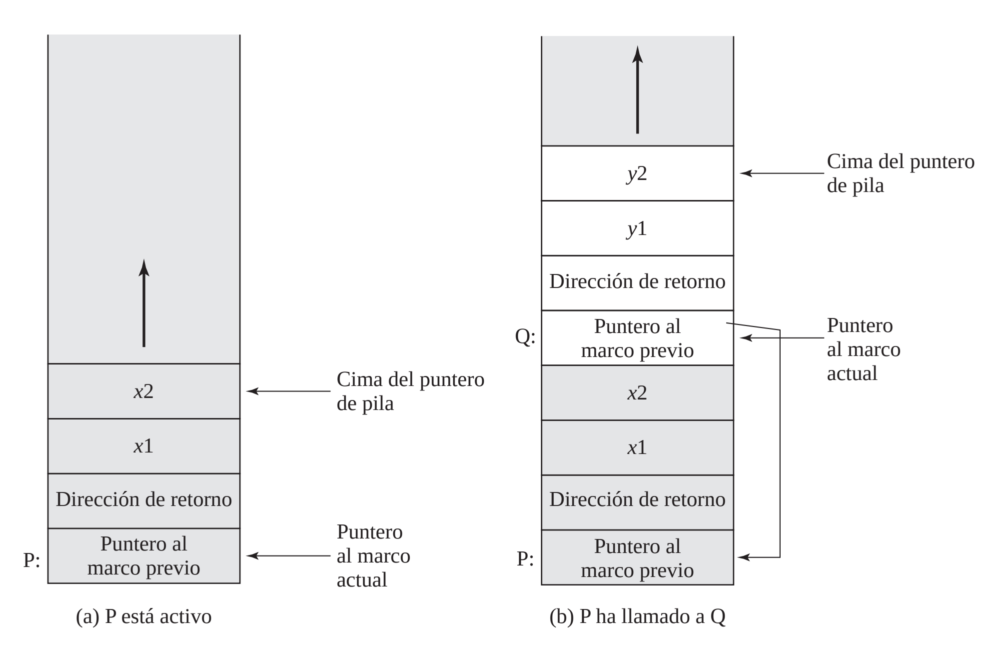


Figura 1.28. Crecimiento del marco de pila utilizando los procedimientos de ejemplo P y Q.

de las cuales se inicia con el valor pasado como parámetro desde P. La otra variable local, $y2$, es simplemente una variable local utilizada por Q en sus cálculos. En el siguiente apartado se analiza la necesidad de incluir las variables locales en el marco de pila.

PROCEDIMIENTOS REENTRANTES

Un concepto útil, particularmente en un sistema que da soporte a múltiples usuarios al mismo tiempo, es el de los procedimientos reentrantes. Un procedimiento reentrante es aquél en el que una única copia del código del programa se puede compartir por múltiples usuarios durante el mismo periodo de tiempo. El carácter reentrante de un procedimiento tiene dos aspectos fundamentales: el código del programa no puede modificarse a sí mismo y los datos locales de cada usuario deben almacenarse separadamente. Un procedimiento reentrante puede ser interrumpido e invocado por el programa que causó la interrupción y, a pesar de ello, ejecutarse correctamente al retornar al procedimiento. En un sistema compartido, el carácter reentrante permite un uso más eficiente de la memoria principal. Se mantiene una copia del código del programa en memoria principal, pero más de una aplicación puede llamar al procedimiento.

Por tanto, un procedimiento reentrante debe tener una parte permanente (las instrucciones que constituyen el procedimiento) y una parte temporal (un puntero de retorno al programa que realizó la llamada, así como espacio para las variables locales usadas por el programa). Cada instancia de la ejecución, llamada activación, de un procedimiento ejecutará el código en la parte permanente pero debe tener su propia copia de los parámetros y las variables locales. La parte temporal asociada con una activación en particular se denomina *registro de activación*.

La manera más conveniente de dar soporte a los procedimientos reentrantes es mediante una pila. Cuando se llama a un procedimiento reentrante, el registro de activación del procedimiento se puede almacenar en la pila. Por tanto, el registro de activación se convierte en parte del marco de pila que se crea en la llamada a procedimiento.

CAPÍTULO 2

Introducción a los sistemas operativos

◆ ◆

Comenzamos nuestro estudio de los sistemas operativos con una breve historia. La historia es interesante y también permite ofrecer una visión general de los principios de los sistemas operativos. La primera sección examina los objetivos y funciones de los sistemas operativos. A continuación, se muestra cómo han evolucionado los sistemas operativos, desde los primitivos sistemas en lotes hasta los sofisticados sistemas multitarea y multiusuario. El resto del capítulo describe la historia y características generales de los dos sistemas operativos que se utilizan como ejemplos a lo largo del libro. Todo el material de este capítulo se cubre en mayor profundidad en el resto de los capítulos.

◆ ◆

2.1. OBJETIVOS Y FUNCIONES DE LOS SISTEMAS OPERATIVOS

Un sistema operativo es un programa que controla la ejecución de aplicaciones y programas y que actúa como interfaz entre las aplicaciones y el hardware del computador. Se puede considerar que un sistema operativo tiene los siguientes tres objetivos:

- **Facilidad de uso.** Un sistema operativo facilita el uso de un computador.
- **Eficiencia.** Un sistema operativo permite que los recursos de un sistema de computación se puedan utilizar de una manera eficiente.
- **Capacidad para evolucionar.** Un sistema operativo se debe construir de tal forma que se puedan desarrollar, probar e introducir nuevas funciones en el sistema sin interferir con su servicio.

Se van a examinar en orden estos tres aspectos de un sistema operativo.

EL SISTEMA OPERATIVO COMO UNA INTERFAZ DE USUARIO/COMPUTADOR

El hardware y software utilizados para proporcionar aplicaciones a los usuarios se pueden ver de forma jerárquica o en capas, tal y como se muestra en la Figura 2.1. El usuario de dichas aplicaciones, es decir, el usuario final, normalmente no se preocupa por los detalles del hardware del computador. Por tanto, el usuario final ve un sistema de computación en términos de un conjunto de aplicaciones. Una aplicación se puede expresar en un lenguaje de programación y normalmente es desarrollada por un programador de aplicaciones. Si un programador tuviera que desarrollar una aplicación como un conjunto de instrucciones en código máquina que se encargaran de controlar completamente el hardware del computador, se enfrentaría a una labor extremadamente compleja. Para facilitar esta tarea, se proporcionan un conjunto de programas de sistema. Algunos de estos programas se conocen como utilidades. Estos programas utilizan frecuentemente funciones que asisten al programador en las fases de creación de programas, gestión de ficheros y control de los dispositivos de E/S. Un programador hará uso de estas utilidades cuando desarrolle una aplicación, y las aplicaciones, invocarán a las utilidades durante su ejecución para llevar a cabo ciertas funciones. El programa de sistema más importante es el sistema operativo. El sistema operativo oculta los detalles del hardware al programador y le proporciona una interfaz apropiada para utilizar el sistema. Actúa como mediador, haciendo más fácil al programador y a la aplicación el acceso y uso de dichas utilidades y servicios.

De forma resumida, el sistema operativo proporciona normalmente servicios en las siguientes áreas:

- **Desarrollo de programas.** El sistema operativo proporciona una variedad de utilidades y servicios, tales como editores y depuradores, para asistir al programador en la creación de los

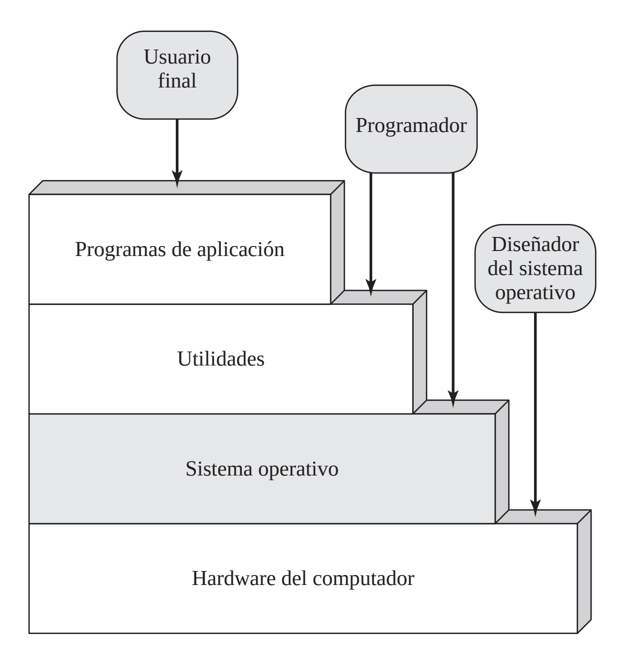


Figura 2.1. Capas y vistas de un sistema de computación.

programas. Normalmente, estos servicios se ofrecen en la forma de utilidades que, aunque no forman parte del núcleo del sistema operativo, se ofrecen con dicho sistema y se conocen como herramientas de desarrollo de programas de aplicación.

- **Ejecución de programas.** Se necesita realizar una serie de pasos para ejecutar un programa. Las instrucciones y los datos se deben cargar en memoria principal. Los dispositivos de E/S y los ficheros se deben inicializar, y otros recursos deben prepararse. Los sistemas operativos realizan estas labores de planificación en nombre del usuario.
- **Acceso a dispositivos de E/S.** Cada dispositivo de E/S requiere su propio conjunto peculiar de instrucciones o señales de control para cada operación. El sistema operativo proporciona una interfaz uniforme que esconde esos detalles de forma que los programadores puedan acceder a dichos dispositivos utilizando lecturas y escrituras sencillas.
- **Acceso controlado a los ficheros.** Para el acceso a los ficheros, el sistema operativo debe reflejar una comprensión detallada no sólo de la naturaleza del dispositivo de E/S (disco, cinta), sino también de la estructura de los datos contenidos en los ficheros del sistema de almacenamiento. Adicionalmente, en el caso de un sistema con múltiples usuarios, el sistema operativo puede proporcionar mecanismos de protección para controlar el acceso a los ficheros.
- **Acceso al sistema.** Para sistemas compartidos o públicos, el sistema operativo controla el acceso al sistema completo y a recursos del sistema específicos. La función de acceso debe proporcionar protección a los recursos y a los datos, evitando el uso no autorizado de los usuarios y resolviendo conflictos en el caso de conflicto de recursos.
- **Detección y respuesta a errores.** Se pueden dar gran variedad de errores durante la ejecución de un sistema de computación. Éstos incluyen errores de hardware internos y externos, tales como un error de memoria, o un fallo en un dispositivo; y diferentes errores software, tales como la división por cero, el intento de acceder a una posición de memoria prohibida o la incapacidad del sistema operativo para conceder la solicitud de una aplicación. En cada caso, el

sistema operativo debe proporcionar una respuesta que elimine la condición de error, suponiendo el menor impacto en las aplicaciones que están en ejecución. La respuesta puede oscilar entre finalizar el programa que causó el error hasta reintentar la operación o simplemente informar del error a la aplicación.

- **Contabilidad.** Un buen sistema operativo recogerá estadísticas de uso de los diferentes recursos y monitorizará parámetros de rendimiento tales como el tiempo de respuesta. En cualquier sistema, esta información es útil para anticipar las necesidades de mejoras futuras y para optimizar el sistema a fin de mejorar su rendimiento. En un sistema multiusuario, esta información se puede utilizar para facturar a los diferentes usuarios.

EL SISTEMA OPERATIVO COMO GESTOR DE RECURSOS

Un computador es un conjunto de recursos que se utilizan para el transporte, almacenamiento y procesamiento de los datos, así como para llevar a cabo el control de estas funciones. El sistema operativo se encarga de gestionar estos recursos.

¿Se puede decir que es el sistema operativo quien controla el transporte, almacenamiento y procesamiento de los datos? Desde un punto de vista, la respuesta es afirmativa: gestionando los recursos del computador, el sistema operativo tiene el control de las funciones básicas del mismo. Pero este control se realiza de una forma curiosa. Normalmente, se habla de un mecanismo de control como algo externo al dispositivo controlado, o al menos como algo que constituye una parte separada o distinta de dicho dispositivo. (Por ejemplo, un sistema de calefacción de una residencia se controla a través de un termostato, que está separado de los aparatos de generación y distribución de calor.) Este no es el caso del sistema operativo, que es un mecanismo de control inusual en dos aspectos:

- Las funciones del sistema operativo actúan de la misma forma que el resto del software; es decir, se trata de un programa o conjunto de programas ejecutados por el procesador.
- El sistema operativo frecuentemente cede el control y depende del procesador para volver a retomarlo.

De hecho, el sistema operativo es un conjunto de programas. Como otros programas, proporciona instrucciones para el procesador. La principal diferencia radica en el objetivo del programa. El sistema operativo dirige al procesador en el uso de los otros recursos del sistema y en la temporización de la ejecución de otros programas. No obstante, para que el procesador pueda realizar esto, el sistema operativo debe dejar paso a la ejecución de otros programas. Por tanto, el sistema operativo deja el control para que el procesador pueda realizar trabajo «útil» y de nuevo retoma el control para permitir al procesador que realice la siguiente pieza de trabajo. Los mecanismos que se utilizan para llevar a cabo esto quedarán más claros a lo largo del capítulo.

La Figura 2.2 muestra los principales recursos gestionados por el sistema operativo. Una porción del sistema operativo se encuentra en la memoria principal. Esto incluye el ***kernel***, o **núcleo**, que contiene las funciones del sistema operativo más frecuentemente utilizadas y, en cierto momento, otras porciones del sistema operativo actualmente en uso. El resto de la memoria principal contiene programas y datos de usuario. La asignación de este recurso (memoria principal) es controlada de forma conjunta por el sistema operativo y el hardware de gestión de memoria del procesador, como se verá. El sistema operativo decide cuándo un programa en ejecución puede utilizar un dispositivo de E/S y controla el acceso y uso de los ficheros. El procesador es también un recurso, y el sistema operativo debe determinar cuánto tiempo de procesador debe asignarse a la ejecución de un programa de usuario particular. En el caso de un sistema multiprocesador, esta decisión debe ser tomada por todos los procesadores.

Sistema de computación

Memoria

Sofware del sistema operativo

Programas y datos

Controlador de E/S

Controlador de E/S

Controlador de E/S

Procesador

Procesador

Dispositivos de E/S

Impresoras, teclados, cámaras digitales, etc.

Almacenamiento

OS

Programas

Datos

Figura 2.2. El sistema operativo como gestor de recursos.

FACILIDAD DE EVOLUCIÓN DE UN SISTEMA OPERATIVO

Un sistema operativo importante debe evolucionar en el tiempo por las siguientes razones:

- **Actualizaciones de hardware más nuevos tipos de hardware.** Por ejemplo, las primeras versiones de los sistemas operativos UNIX e IBM OS/2 no empleaban un mecanismo de paginado porque ejecutaban en máquinas sin hardware de paginación. La paginación se presenta brevemente, más adelante en este capítulo, y se discute detalladamente en el Capítulo 7. Versiones más recientes de estos sistemas operativos han cambiado esta faceta para explotar las capacidades de paginación. Además, el uso de terminales gráficos y en modo página en lugar de terminales de línea también afecta al diseño de los sistemas operativos. Por ejemplo, un terminal gráfico normalmente permite al usuario ver varias aplicaciones al mismo tiempo a través del uso de «ventanas» en la pantalla. Esto requiere una gestión más sofisticada por parte del sistema operativo.
- **Nuevos servicios.** En respuesta a la demanda del usuario o en respuesta a las necesidades de los gestores de sistema, el sistema operativo debe ofrecer nuevos servicios. Por ejemplo, si es difícil mantener un buen rendimiento con las herramientas existentes, se pueden añadir al sistema operativo nuevas herramientas de medida y control. Como segundo ejemplo, la mayoría de las aplicaciones requieren el uso de ventanas en la pantalla. Esta característica requiere actualizaciones importantes en el sistema operativo si éste no soporta ventanas.
- **Resolución de fallos.** Cualquier sistema operativo tiene fallos. Estos fallos se descubren con el transcurso del tiempo y se resuelven. Por supuesto, esto implica la introducción de nuevos fallos.

La necesidad de cambiar regularmente un sistema operativo introduce ciertos requisitos en su diseño. Un hecho obvio es que el sistema debe tener un diseño modular, con interfaces entre los módulos claramente definidas, y que debe estar bien documentado. Para programas grandes, tal como el típico sistema operativo contemporáneo, llevar a cabo una modularización sencilla no es adecuado [DENN80a]. Se detallará este tema más adelante en el capítulo.

2.2. LA EVOLUCIÓN DE LOS SISTEMAS OPERATIVOS

Para comprender los requisitos claves de un sistema operativo y el significado de las principales características de un sistema operativo contemporáneo, es útil considerar la evolución de los sistemas operativos a lo largo de los años.

PROCESAMIENTO SERIE

Con los primeros computadores, desde finales de los años 40 hasta mediados de los años 50, el programador interaccionaba directamente con el hardware del computador; no existía ningún sistema operativo. Estas máquinas eran utilizadas desde una consola que contenía luces, interruptores, algún dispositivo de entrada y una impresora. Los programas en código máquina se cargaban a través del dispositivo de entrada (por ejemplo, un lector de tarjetas). Si un error provocaba la parada del programa, las luces indicaban la condición de error. El programador podía entonces examinar los registros del procesador y la memoria principal para determinar la causa de error. Si el programa terminaba de forma normal, la salida aparecía en la impresora.

Estos sistemas iniciales presentaban dos problemas principales:

- **Planificación.** La mayoría de las instalaciones utilizaban una plantilla impresa para reservar tiempo de máquina. Típicamente, un usuario podía solicitar un bloque de tiempo en múltiplos de media hora aproximadamente. Un usuario podía obtener una hora y terminar en 45 minutos; esto implicaba malgastar tiempo de procesamiento del computador. Por otro lado, el usuario podía tener problemas, si no finalizaba en el tiempo asignado y era forzado a terminar antes de resolver el problema.
- **Tiempo de configuración.** Un único programa, denominado **trabajo**, podía implicar la carga en memoria del compilador y del programa en lenguaje de alto nivel (programa en código fuente) y a continuación la carga y el enlace del programa objeto y las funciones comunes. Cada uno de estos pasos podían suponer montar y desmontar cintas o configurar tarjetas. Si ocurría un error, el desgraciado usuario normalmente tenía que volver al comienzo de la secuencia de configuración. Por tanto, se utilizaba una cantidad considerable de tiempo en configurar el programa que se iba a ejecutar.

Este modo de operación puede denominarse procesamiento serie, para reflejar el hecho de que los usuarios acceden al computador en serie. A lo largo del tiempo, se han desarrollado varias herramientas de software de sistemas con el fin de realizar el procesamiento serie más eficiente. Estas herramientas incluyen bibliotecas de funciones comunes, enlazadores, cargadores, depuradores, y rutinas de gestión de E/S disponibles como software común para todos los usuarios.

SISTEMAS EN LOTES SENCILLOS

Las primeras máquinas eran muy caras, y por tanto, era importante maximizar su utilización. El tiempo malgastado en la planificación y configuración de los trabajos era inaceptable.

Para mejorar su utilización, se desarrolló el concepto de sistema operativo en lotes. Parece ser que el primer sistema operativo en lotes (y el primer sistema operativo de cualquier clase) fue desarrollado a mediados de los años 50 por General Motors para el uso de un IBM 701 [WEIZ81]. El concepto fue subsecuentemente refinado e implementado en el IBM 704 por un número de clientes de IBM. A principios de los años 60, un número de vendedores había desarrollado sistemas operativos en lote para sus sistemas de computación. IBSYS, el sistema operativo de IBM para los computadores 7090/7094, es particularmente notable por su gran influencia en otros sistemas.

La idea central bajo el esquema de procesamiento en lotes sencillo es el uso de una pieza de software denomina **monitor**. Con este tipo de sistema operativo, el usuario no tiene que acceder directamente a la máquina. En su lugar, el usuario envía un trabajo a través de una tarjeta o cinta al operador del computador, que crea un sistema por lotes con todos los trabajos enviados y coloca la secuencia de trabajos en el dispositivo de entrada, para que lo utilice el monitor. Cuando un programa finaliza su procesamiento, devuelve el control al monitor, punto en el cual dicho monitor comienza la carga del siguiente programa.

Para comprender cómo funciona este esquema, se puede analizar desde dos puntos de vista: el del monitor y el del procesador.

- **Punto de vista del monitor.** El monitor controla la secuencia de eventos. Para ello, una gran parte del monitor debe estar siempre en memoria principal y disponible para la ejecución (Figura 2.3). Esta porción del monitor se denomina **monitor residente**. El resto del monitor está formado por un conjunto de utilidades y funciones comunes que se cargan como subrutinas en el programa de usuario, al comienzo de cualquier trabajo que las requiera. El monitor lee de uno en uno los trabajos desde el dispositivo de entrada (normalmente un lector de tarjetas o dispositivo de cinta magnética). Una vez leído el dispositivo, el trabajo actual se coloca en el área de programa de usuario, y se le pasa el control. Cuando el trabajo se ha completado, devuelve el control al monitor, que inmediatamente lee el siguiente trabajo. Los resultados de cada trabajo se envían a un dispositivo de salida, (por ejemplo, una impresora), para entregárselo al usuario.

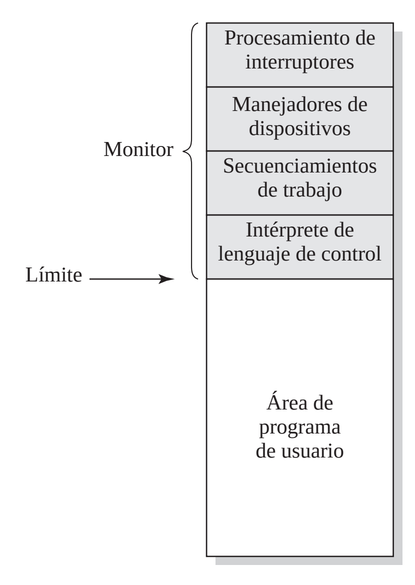
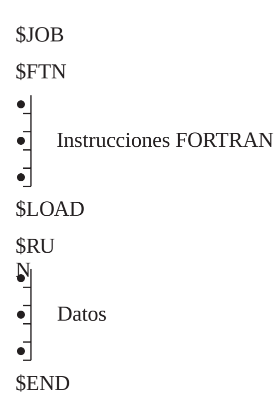


Figura 2.3. Disposición de memoria de un monitor residente.

- **Punto de vista del procesador.** En un cierto punto, el procesador ejecuta instrucciones de la zona de memoria principal que contiene el monitor. Estas instrucciones provocan que se lea el siguiente trabajo y se almacene en otra zona de memoria principal. Una vez que el trabajo se ha leído, el procesador encontrará una instrucción de salto en el monitor que le indica al procesador que continúe la ejecución al inicio del programa de usuario. El procesador entonces ejecutará las instrucciones del programa usuario hasta que encuentre una condición de finalización o de error. Cualquiera de estas condiciones hace que el procesador ejecute la siguiente instrucción del programa monitor. Por tanto, la frase «se pasa el control al trabajo» simplemente significa que el procesador leerá y ejecutará instrucciones del programa de usuario, y la frase «se devuelve el control al monitor» indica que el procesador leerá y ejecutará instrucciones del programa monitor.

El monitor realiza una función de planificación: en una cola se sitúa un lote de trabajos, y los trabajos se ejecutan lo más rápidamente posible, sin ninguna clase de tiempo ocioso entre medias. Además, el monitor mejora el tiempo de configuración de los trabajos. Con cada uno de los trabajos, se incluye un conjunto de instrucciones en algún formato primitivo de **lenguaje de control de trabajos** (*Job Control Language*, JCL). Se trata de un tipo especial de lenguaje de programación utilizado para dotar de instrucciones al monitor. Un ejemplo sencillo consiste en un usuario enviando un programa escrito en el lenguaje de programación FORTRAN más algunos datos que serán utilizados por el programa. Además del código en FORTRAN y las líneas de datos, el trabajo incluye instrucciones de control del trabajo, que se representan mediante líneas que comienzan mediante el símbolo `$´. El formato general del trabajo tiene el siguiente aspecto:



Para ejecutar este trabajo, el monitor lee la línea $FTN y carga el compilador apropiado de su sistema de almacenamiento (normalmente una cinta). El compilador traduce el programa de usuario en código objeto, el cual se almacena en memoria en el sistema de almacenamiento. Si se almacena en memoria, la operación se denomina «compilar, cargar, y ejecutar». En el caso de que se almacene en una cinta, se necesita utilizar la instrucción $LOAD. El monitor lee esta instrucción y recupera el control después de la operación de compilación. El monitor invoca al cargador, que carga el programa objeto en memoria (en el lugar del compilador) y le transfiere el control. De esta forma, se puede compartir un gran segmento de memoria principal entre diferentes subsistemas, aunque sólo uno de ellos se puede ejecutar en un momento determinado.

Durante la ejecución del programa de usuario, cualquier instrucción de entrada implica la lectura de una línea de datos. La instrucción de entrada del programa de usuario supone la invocación de una rutina de entrada, que forma parte del sistema operativo. La rutina de entrada comprueba que el pro-

grama no lea accidentalmente una línea JCL. Si esto sucede, se genera un error y se transfiere el control al monitor. Al finalizar el trabajo de usuario, el monitor analizará todas las líneas de entrada hasta que encuentra la siguiente instrucción JCL. De esta forma, el sistema queda protegido frente a un programa con excesivas o escasas líneas de datos.

El monitor, o sistema operativo en lotes, es simplemente un programa. Éste confía en la habilidad del procesador para cargar instrucciones de diferentes porciones de la memoria principal que de forma alternativa le permiten tomar y abandonar el control. Otras características hardware son también deseables:

- **Protección de memoria.** Durante la ejecución del programa de usuario, éste no debe alterar el área de memoria que contiene el monitor. Si esto ocurriera, el hardware del procesador debe detectar un error y transferir el control al monitor. El monitor entonces abortará el trabajo, imprimirá un mensaje de error y cargará el siguiente trabajo.
- **Temporizador.** Se utiliza un temporizador para evitar que un único trabajo monopolice el sistema. Se activa el temporizador al comienzo de cada trabajo. Si el temporizador expira, se para el programa de usuario, y se devuelve el control al monitor.
- **Instrucciones privilegiadas.** Ciertas instrucciones a nivel de máquina se denominan privilegiadas y sólo las puede ejecutar el monitor. Si el procesador encuentra estas instrucciones mientras ejecuta un programa de usuario, se produce un error provocando que el control se transfiera al monitor. Entre las instrucciones privilegiadas se encuentran las instrucciones de E/S, que permiten que el monitor tome control de los dispositivos de E/S. Esto evita, por ejemplo, que un programa de usuario de forma accidental lea instrucciones de control de trabajos del siguiente trabajo. Si un programa de usuario desea realizar operaciones de E/S, debe solicitar al monitor que realice las operaciones por él.
- **Interrupciones.** Los modelos de computadores iniciales no tenían esta capacidad. Esta característica proporciona al sistema operativo más flexibilidad para dejar y retomar el control desde los programas de usuario.

Ciertas consideraciones sobre la protección de memoria y las instrucciones privilegiadas llevan al concepto de modos de operación. Un programa de usuario ejecuta en **modo usuario**, en el cual los usuarios no pueden acceder a ciertas áreas de memoria y no puede ejecutar ciertas instrucciones. El monitor ejecuta en modo sistema, o lo que se denomina **modo núcleo**, en el cual se pueden ejecutar instrucciones privilegiadas y se puede acceder a áreas de memoria protegidas.

Por supuesto, se puede construir un sistema operativo sin estas características. Pero los fabricantes de computadores rápidamente se dieron cuenta de que los resultados no eran buenos, y de este modo, se construyeron sistemas operativos en lotes primitivos con estas características hardware.

Con un sistema operativo en lotes, el tiempo de máquina alterna la ejecución de programas de usuario y la ejecución del monitor. Esto implica dos sacrificios: el monitor utiliza parte de la memoria principal y consume parte del tiempo de máquina. Ambas situaciones implican una sobrecarga. A pesar de esta sobrecarga, el sistema en lotes simple mejora la utilización del computador.

SISTEMAS EN LOTES MULTIPROGRAMADOS

El procesador se encuentra frecuentemente ocioso, incluso con el secuenciamiento de trabajos automático que proporciona un sistema operativo en lotes simple. El problema consiste en que los dispositivos de E/S son lentos comparados con el procesador. La Figura 2.4 detalla un cálculo representativo de este hecho, que corresponde a un programa que procesa un fichero con registros y

realiza de media 100 instrucciones máquina por registro. En este ejemplo, el computador malgasta aproximadamente el 96% de su tiempo esperando a que los dispositivos de E/S terminen de transferir datos a y desde el fichero. La Figura 2.5a muestra esta situación, donde existe un único programa, lo que se denomina monoprogramación. El procesador ejecuta durante cierto tiempo hasta que alcanza una instrucción de E/S. Entonces debe esperar que la instrucción de E/S concluya antes de continuar.

Leer un registro del fichero	15 μs
Ejecutar 100 instrucciones	1 μs
Escribir un registro al fichero	15 μs
TOTAL	31 μs

Porcentaje de utilización de la CPU = $\frac{1}{31}$ = 0,032 = 3,2%

Figura 2.4. Ejemplo de utilización del sistema.

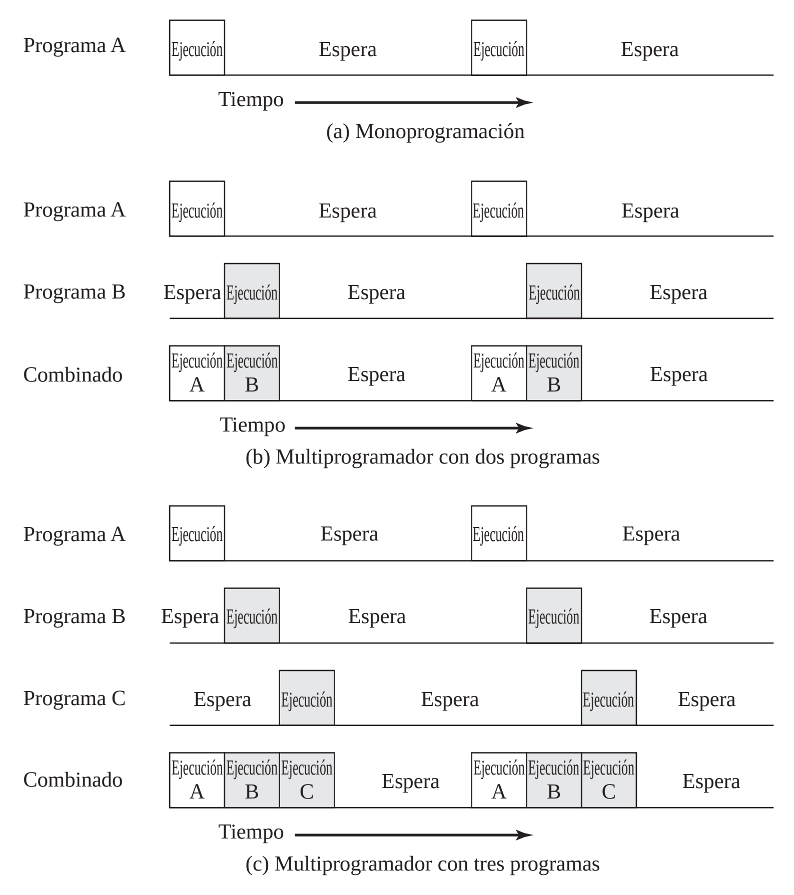


Figura 2.5. Ejemplo de multiprogramación.

Esta ineficiencia puede evitarse. Se sabe que existe suficiente memoria para contener al sistema operativo (monitor residente) y un programa de usuario. Supóngase que hay espacio para el sistema operativo y dos programas de usuario. Cuando un trabajo necesita esperar por la E/S, se puede asignar el procesador al otro trabajo, que probablemente no esté esperando por una operación de E/S (Figura 2.5b). Más aún, se puede expandir la memoria para que albergue tres, cuatro o más programas y pueda haber multiplexación entre todos ellos (Figura 2.5c). Este enfoque se conoce como **multiprogramación** o **multitarea**. Es el tema central de los sistemas operativos modernos.

Para mostrar los beneficios de la multiprogramación, se describe un ejemplo sencillo. Sea un computador con 250 Mbytes de memoria disponible (sin utilizar por el sistema operativo), un disco, un terminal y una impresora. Se envían simultáneamente a ejecución tres programas TRABAJO1, TRABAJO2 y TRABAJO3, con las características listadas en la Tabla 2.1. Se asumen requisitos mínimos de procesador para los trabajos TRABAJO1 y TRABAJO3, así como uso continuo de impresora y disco por parte del trabajo TRABAJO3. En un entorno por lotes sencillo, estos trabajos se ejecutarán en secuencia. Por tanto, el trabajo TRABAJO1 se completará en 5 minutos. El trabajo TRABAJO2 esperará estos 5 minutos y a continuación se ejecutará, terminando 15 minutos después. El trabajo TRABAJO3 esperará estos 20 minutos y se completará 30 minutos después de haberse enviado. La media de utilización de recursos, productividad y tiempos de respuestas se muestran en la columna de monoprogramación de la Tabla 2.2. La utilización de cada dispositivo se ilustra en la Figura 2.6a. Es evidente que existe una infrautilización de todos los recursos cuando se compara respecto al periodo de 30 minutos requerido.

Tabla 2.1. Atributos de ejecución de ejemplos de programas.

	TRABAJO 1	**TRABAJO 2**	**TRABAJO 3**
Tipo de trabajo	Computación pesada	Gran cantidad de E/S	Gran cantidad de E/S
Duración	5 minutos	15 minutos	10 minutos
Memoria requerida	50 M	100 M	75 M
¿Necesita disco?	No	No	Sí
¿Necesita terminal?	No	Sí	No
¿Necesita impresora?	No	No	Sí

Tabla 2.2. Efectos de la utilización de recursos sobre la multiprogramación.

	Monoprogramación	**Multiprogramación**
Uso de procesador	20%	40%
Uso de memoria	33%	67%
Uso de disco	33%	67%
Uso de impresora	33%	67%
Tiempo transcurrido	30 minutos	15 minutos
Productividad	6 trabajos/hora	12 trabajos/hora
Tiempo de respuesta medio	18 minutos	10 minutos

Ahora supóngase que los trabajos se ejecutan concurrentemente bajo un sistema operativo multiprogramado. Debido a que hay poco conflicto entre los trabajos, todos pueden ejecutar casi en el mínimo tiempo mientras coexisten con los otros en el computador (asumiendo que se asigne a los trabajos TRABAJO2 y TRABAJO3 suficiente tiempo de procesador para mantener sus operaciones de entrada y salida activas). El trabajo TRABAJO1 todavía requerirá 5 minutos para completarse, pero al final de este tiempo, TRABAJO2 habrá completado un tercio de su trabajo y TRABAJO3 la mitad. Los tres trabajos habrán finalizado en 15 minutos. La mejora es evidente al examinar la columna de multiprogramación de la Tabla 2.2, obtenido del histograma mostrado en la Figura 2.6b.

Del mismo modo que un sistema en lotes simple, un sistema en lotes multiprogramado también debe basarse en ciertas características hardware del computador. La característica adicional más notable que es útil para la multiprogramación es el hardware que soporta las interrupciones de E/S y DMA (*Direct Memory Access*: acceso directo a memoria). Con la E/S gestionada a través de interrupciones o DMA, el procesador puede solicitar un mandato de E/S para un trabajo y continuar con la ejecución de otro trabajo mientras el controlador del dispositivo gestiona dicha operación de E/S. Cuando esta última operación finaliza, el procesador es interrumpido y se pasa el control a un programa de tratamiento de interrupciones del sistema operativo. Entonces, el sistema operativo pasará el control a otro trabajo.

Los sistemas operativos multiprogramados son bastante sofisticados, comparados con los sistemas **monoprogramados**. Para tener varios trabajos listos para ejecutar, éstos deben guardarse en memoria principal, requiriendo alguna forma de **gestión de memoria**. Adicionalmente, si varios trabajos están listos para su ejecución, el procesador debe decidir cuál de ellos ejecutar; esta decisión requiere un algoritmo para planificación. Estos conceptos se discuten más adelante en este capítulo.

SISTEMAS DE TIEMPO COMPARTIDO

Con el uso de la multiprogramación, el procesamiento en lotes puede ser bastante eficiente. Sin embargo, para muchos trabajos, es deseable proporcionar un modo en el cual el usuario interaccione directamente con el computador. De hecho, para algunos trabajos, tal como el procesamiento de transacciones, un modo interactivo es esencial.

Hoy en día, los computadores personales dedicados o estaciones de trabajo pueden cumplir, y frecuentemente lo hacen, los requisitos que necesita una utilidad de computación interactiva. Esta opción no estuvo disponible hasta los años 60, cuando la mayoría de los computadores eran grandes y costosos. En su lugar, se desarrolló el concepto de tiempo compartido.

Del mismo modo que la multiprogramación permite al procesador gestionar múltiples trabajos en lotes en un determinado tiempo, la multiprogramación también se puede utilizar para gestionar múltiples trabajos interactivos. En este último caso, la técnica se denomina **tiempo compartido**, porque se comparte el tiempo de procesador entre múltiples usuarios. En un sistema de tiempo compartido, múltiples usuarios acceden simultáneamente al sistema a través de terminales, siendo el sistema operativo el encargado de entrelazar la ejecución de cada programa de usuario en pequeños intervalos de tiempo o cuantos de computación. Por tanto, si hay n usuarios activos solicitando un servicio a la vez, cada usuario sólo verá en media $1/n$ de la capacidad de computación efectiva, sin contar la sobrecarga introducida por el sistema operativo. Sin embargo, dado el tiempo de reacción relativamente lento de los humanos, el tiempo de respuesta de un sistema diseñado adecuadamente debería ser similar al de un computador dedicado.

Ambos tipos de procesamiento, en lotes y tiempo compartido, utilizan multiprogramación. Las diferencias más importantes se listan en la Tabla 2.3.

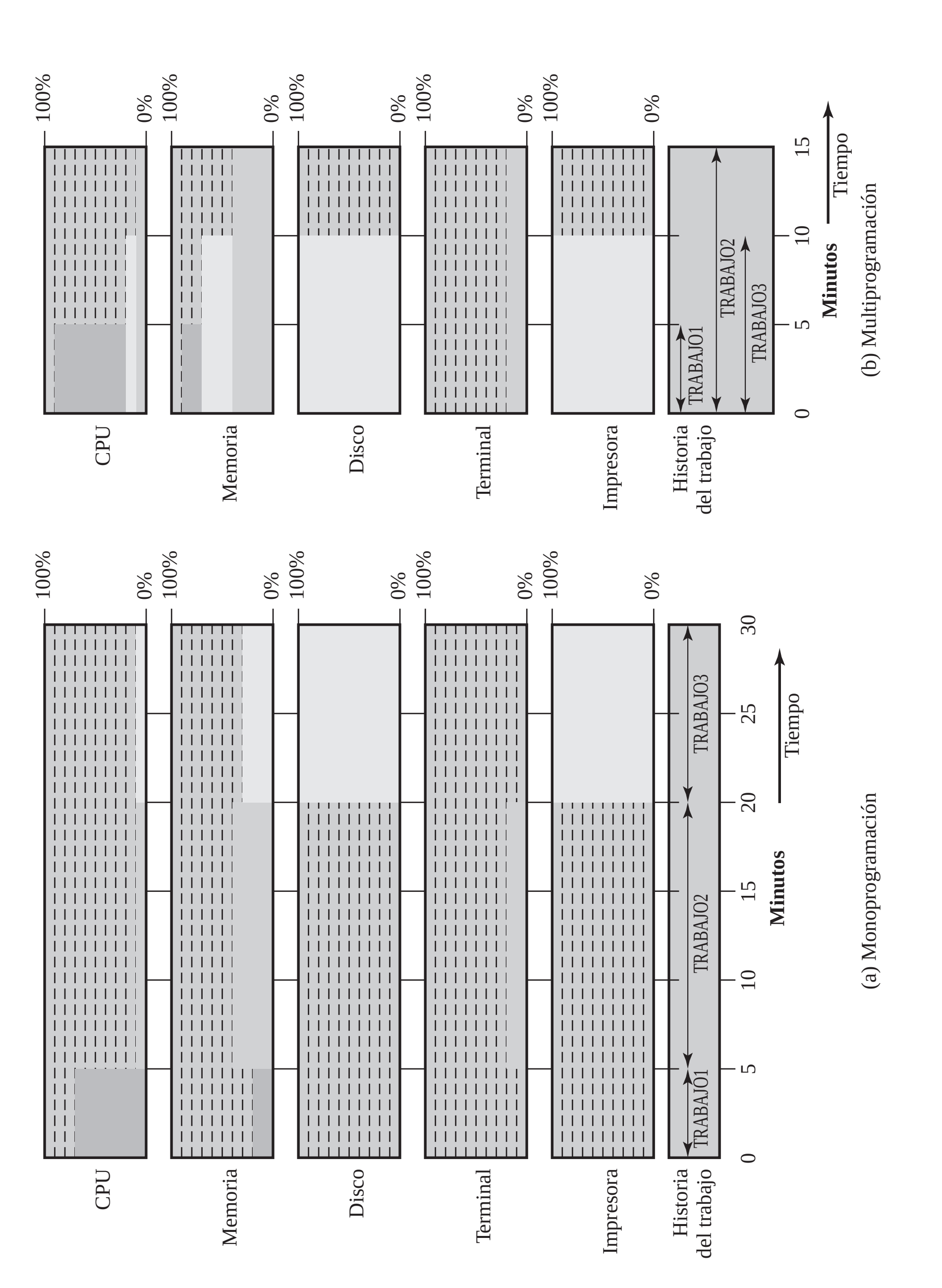


Figura 2.6. Histogramas de utilización.

Tabla 2.3. Multiprogramación en lotes frente a tiempo compartido.

	Multiprogramación en lotes	**Tiempo compartido**
Objetivo principal	Maximizar el uso del procesador	Minimizar el tiempo de respuesta
Fuente de directivas al sistema operativo	Mandatos del lenguaje de control de trabajos proporcionados por el trabajo	Mandatos introducidos al terminal

Uno de los primeros sistemas operativos de tiempo compartido desarrollados fue el sistema CTSS (*Compatible Time-Sharing System*) [CORB62], desarrollado en el MIT por un grupo conocido como Proyecto MAC (*Machine-Aided Cognition*, o *Multiple-Access Computers*). El sistema fue inicialmente desarrollado para el IBM 709 en 1961 y más tarde transferido al IBM 7094.

Comparado con sistemas posteriores, CTSS es primitivo. El sistema ejecutó en una máquina con memoria principal con 32.000 palabras de 36 bits, con el monitor residente ocupando 5000 palabras. Cuando el control se asignaba a un usuario interactivo, el programa de usuario y los datos se cargaban en las restantes 27.000 palabras de memoria principal. Para arrancar, un programa siempre se cargaba al comienzo de la palabra 5000; esto simplificaba tanto el monitor como la gestión de memoria. Un reloj del sistema generaba una interrupción cada 0,2 segundos aproximadamente. En cada interrupción de reloj, el sistema operativo retomaba el control y podía asignar el procesador a otro usuario. Por tanto, a intervalos regulares de tiempo, el usuario actual podría ser desalojado y otro usuario puesto a ejecutar. Para preservar el estado del programa de usuario antiguo, los programas de usuario y los datos se escriben en el disco antes de que se lean los nuevos programas de usuario y nuevos datos. Posteriormente, el código y los datos del programa de usuario antiguo se restauran en memoria principal cuando dicho programa vuelve a ser planificado.

Para minimizar el tráfico de disco, la memoria de usuario sólo es escrita a disco cuando el programa entrante la sobreescribe. Este principio queda ilustrado en la Figura 2.7. Sean cuatro usuarios interactivos con los siguiente requisitos de memoria:

- TRABAJO1: 15.000
- TRABAJO2: 20.000
- TRABAJO3: 5000
- TRABAJO4: 10.000

Inicialmente, el monitor carga el trabajo TRABAJO1 y le transfiere control (a). Después, el monitor decide transferir el control al trabajo TRABAJO2. Debido a que el TRABAJO2 requiere más memoria que el TRABAJO1, se debe escribir primero el TRABAJO1 en disco, y a continuación debe cargarse el TRABAJO2 (b). A continuación, se debe cargar el TRABAJO3 para ejecutarse. Sin embargo, debido a que el TRABAJO3 es más pequeño que el TRABAJO2, una porción de este último queda en memoria, reduciendo el tiempo de escritura de disco (c). Posteriormente, el monitor decide transferir el control de nuevo al TRABAJO1. Una porción adicional de TRABAJO2 debe escribirse en disco cuando se carga de nuevo el TRABAJO1 en memoria (d). Cuando se carga el TRABAJO4, parte del trabajo TRABAJO1 y la porción de TRABAJO2 permanecen en memoria (e). En este punto, si cualquiera de estos trabajos (TRABAJO1 o TRABAJO2) son activados, sólo se requiere una carga parcial. En este ejemplo, es el TRABAJO2 el que ejecuta de nuevo. Esto requiere que el TRABAJO4 y la porción residente de el TRABAJO1 se escriban en el disco y la parte que falta de el TRABAJO2 se lea (f).

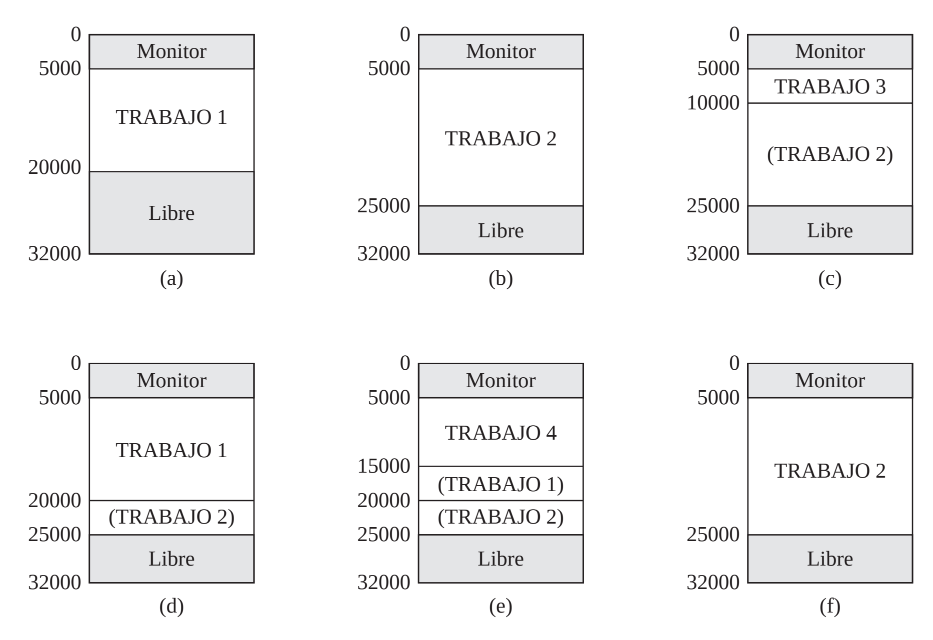


Figura 2.7. Operación del CTSS.

La técnica utilizada por CTSS es primitiva comparada con las técnicas de tiempo compartido actuales, pero funcionaba. Era extremadamente sencilla, lo que minimizaba el tamaño del monitor. Debido a que un trabajo siempre se cargaba en la misma dirección de memoria, no había necesidad de utilizar técnicas de reubicación en tiempo de carga (que se discutirán más adelante). El hecho de sólo escribir en disco cuando es necesario, minimiza la actividad del disco. Ejecutando sobre el 7094, CTSS permitía un número máximo de 32 usuarios.

La compartición de tiempo y la multiprogramación implican nuevos problemas para el sistema operativo. Si existen múltiples trabajos en memoria, éstos deben protegerse para evitar que interfieran entre sí, por ejemplo, a través de la modificación de los datos de los mismos. Con múltiples usuarios interactivos, el sistema de ficheros debe ser protegido, de forma que sólo usuarios autorizados tengan acceso a un fichero particular. También debe gestionarse los conflictos entre los recursos, tal como impresoras y dispositivos de almacenamiento masivo. Éstos y otros problemas, con sus posibles soluciones, se describirán a lo largo de este libro.

2.3. PRINCIPALES LOGROS

Los sistemas operativos se encuentran entre las piezas de software más complejas jamás desarrolladas. Esto refleja el reto de intentar resolver la dificultad de alcanzar determinados objetivos, algunas veces conflictivos, de conveniencia, eficiencia y capacidad de evolución. [DENN80a] propone cinco principales avances teóricos en el desarrollo de los sistemas operativos:

- Procesos.
- Gestión de memoria.
- Protección y seguridad de la información.
- Planificación y gestión de los recursos.
- Estructura del sistema.

Cada avance se caracteriza por principios, o abstracciones, que se han desarrollado para resolver problemas prácticos. Tomadas de forma conjunta, estas cinco áreas incluyen la mayoría de los aspectos clave de diseño e implementación de los sistemas operativos modernos. La breve revisión de estas cinco áreas en esta sección sirve como una visión global de gran parte del resto del libro.

PROCESOS

El concepto de proceso es fundamental en la estructura de los sistemas operativos. Este término fue utilizado por primera vez por los diseñadores del sistema Multics en los años 60 [DALE68]. Es un término un poco más general que el de trabajo. Se han dado muchas definiciones del término *proceso*, incluyendo:

- Un programa en ejecución.
- Una instancia de un programa ejecutándose en un computador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad caracterizada por un solo hilo secuencial de ejecución, un estado actual, y un conjunto de recursos del sistema asociados.

Este concepto se aclarará a lo largo del texto.

Tres líneas principales de desarrollo del sistema de computación crearon problemas de temporización y sincronización que contribuyeron al desarrollo del concepto de proceso: operación en lotes multiprogramados, tiempo compartido, y sistemas de transacciones de tiempo real. Como ya se ha visto, la multiprogramación se diseñó para permitir el uso simultáneo del procesador y los dispositivos de E/S, incluyendo los dispositivos de almacenamiento, para alcanzar la máxima eficiencia. El mecanismo clave es éste: en respuesta a las señales que indican la finalización de las transacciones de E/S, el procesador es planificado para los diferentes programas que residen en memoria principal.

Una segunda línea de desarrollo fue el tiempo compartido de propósito general. En este caso, el objetivo clave de diseño es responder a las necesidades del usuario y, debido a razones económicas, ser capaz de soportar muchos usuarios simultáneamente. Estos objetivos son compatibles debido al tiempo de reacción relativamente lento del usuario. Por ejemplo, si un usuario típico necesita una media de 2 segundos de tiempo de procesamiento por minuto, entonces una cantidad de 30 usuarios aproximadamente podría compartir el mismo sistema sin interferencias notables. Por supuesto, la sobrecarga del sistema debe tenerse en cuenta para realizar estos cálculos.

Otra línea importante de desarrollo han sido los sistemas de procesamiento de transacciones de tiempo real. En este caso, un cierto número de usuarios realizan consultas o actualizaciones sobre una base de datos. Un ejemplo es un sistema de reserva para una compañía aérea. La principal diferencia entre el sistema de procesamiento de transacciones y el sistema de tiempo real es que el primero está limitado a una o unas pocas aplicaciones, mientras que los usuarios de un sistema de tiempo real pueden estar comprometidos en el desarrollo de programas, la ejecución de trabajos, y el uso de varias aplicaciones. En ambos casos, el tiempo de respuesta del sistema es impresionante.

La principal herramienta disponible para programadores de sistema para el desarrollo de la inicial multiprogramación y los sistemas interactivos multiusuario fue la interrupción. Cualquier trabajo podía suspender su actividad por la ocurrencia de un evento definido, tal como la finalización de una operación de E/S. El procesador guardaría alguna forma de contexto (por ejemplo, el contador de programa y otros registros) y saltaría a una rutina de tratamiento de interrupciones, que determinaría la naturaleza de la interrupción, procesaría la interrupción, y después continuaría el procesamiento de usuario con el trabajo interrumpido o algún otro trabajo.

El diseño del software del sistema para coordinar estas diversas actividades resultó ser notablemente difícil. Con la progresión simultánea de muchos trabajos, cada uno de los cuales suponía la realización de numerosos pasos para su ejecución secuencial, era imposible analizar todas las posibles combinaciones de secuencias de eventos. Con la ausencia de algún método sistemático de coordinación y cooperación entre las actividades, los programadores acudían a métodos *«ad hoc»* basados en la comprensión del entorno que el sistema operativo tenía que controlar. Estos esfuerzos eran vulnerables frente a errores de programación sutiles, cuyos efectos sólo podían observarse cuando ciertas extrañas secuencias de acciones ocurrían. Estos errores eran difíciles de diagnosticar, porque necesitaban distinguirse de los errores software y hardware de las aplicaciones. Incluso cuando se detectaba el error, era difícil determinar la causa, porque las condiciones precisas bajo las cuales el error aparecía, eran difíciles de reproducir. En términos generales, existen cuatro causas principales de dichos errores [DEBB80a]:

- **Inapropiada sincronización.** Es frecuente el hecho de que una rutina se suspenda esperando por algún evento en el sistema. Por ejemplo, un programa que inicia una lectura de E/S debe esperar hasta que los datos estén disponibles en un *buffer* antes de proceder. En este caso, se necesita una señal procedente de otra rutina. El diseño inapropiado del mecanismo de señalización puede provocar que las señales se pierdan o se reciban señales duplicadas.
- **Violación de la exclusión mutua.** Frecuentemente, más de un programa o usuario intentan hacer uso de recursos compartidos simultáneamente. Por ejemplo, dos usuarios podrían intentar editar el mismo fichero a la vez. Si estos accesos no se controlan, podría ocurrir un error. Debe existir algún tipo de mecanismo de exclusión mutua que permita que sólo una rutina en un momento determinado actualice un fichero. Es difícil verificar que la implementación de la exclusión mutua es correcta en todas las posibles secuencias de eventos.
- **Operación no determinista de un programa.** Los resultados de un programa particular normalmente dependen sólo de la entrada a dicho programa y no de las actividades de otro programa en un sistema compartido. Pero cuando los programas comparten memoria, y sus ejecuciones son entrelazadas por el procesador, podrían interferir entre ellos, sobreescribiendo zonas de memoria comunes de una forma impredecible. Por tanto, el orden en el que diversos programas se planifican puede afectar a la salida de cualquier programa particular.
- **Interbloqueos.** Es posible que dos o más programas se queden bloqueados esperándose entre sí. Por ejemplo, dos programas podrían requerir dos dispositivos de E/S para llevar a cabo una determinada operación (por ejemplo, una copia de un disco o una cinta). Uno de los programas ha tomado control de uno de los dispositivos y el otro programa tiene control del otro dispositivo. Cada uno de ellos está esperando a que el otro programa libere el recurso que no poseen. Dicho interbloqueo puede depender de la temporización de la asignación y liberación de recursos.

Lo que se necesita para enfrentarse a estos problemas es una forma sistemática de monitorizar y controlar la ejecución de varios programas en el procesador. El concepto de proceso proporciona los fundamentos. Se puede considerar que un proceso está formado por los siguientes tres componentes:

- Un programa ejecutable.
- Los datos asociados que necesita el programa (variables, espacio de trabajo, *buffers*, etc.).
- El contexto de ejecución del programa.

Este último elemento es esencial. El **contexto de ejecución**, o **estado del proceso**, es el conjunto de datos interno por el cual el sistema operativo es capaz de supervisar y controlar el proceso. Esta información interna está separada del proceso, porque el sistema operativo tiene información a la que

el proceso no puede acceder. El contexto incluye toda la información que el sistema operativo necesita para gestionar el proceso y que el procesador necesita para ejecutar el proceso apropiadamente. El contexto incluye el contenido de diversos registros del procesador, tales como el contador de programa y los registros de datos. También incluye información de uso del sistema operativo, como la prioridad del proceso y si un proceso está esperando por la finalización de un evento de E/S particular.

La Figura 2.8 indica una forma en la cual los procesos pueden gestionarse. Dos procesos, A y B, se encuentran en una porción de memoria principal. Es decir, se ha asignado un bloque de memoria a cada proceso, que contiene el programa, datos e información de contexto. Se incluye a cada proceso en una lista de procesos que construye y mantiene el sistema operativo. La lista de procesos contiene una entrada por cada proceso, e incluye un puntero a la ubicación del bloque de memoria que contiene el proceso. La entrada podría también incluir parte o todo el contexto de ejecución del proceso. El resto del contexto de ejecución es almacenado en otro lugar, tal vez junto al propio proceso (como queda reflejado en la Figura 2.8) o frecuentemente en una región de memoria separada. El registro índice del proceso contiene el índice del proceso que el procesador está actualmente controlando en la lista de procesos. El contador de programa apunta a la siguiente instrucción del proceso que se va a ejecutar. Los registros base y límite definen la región de memoria ocupada por el proceso: el registro base contiene la dirección inicial de la región de memoria y el registro límite el tamaño de la región (en bytes o palabras). El contador de programa y todas las referencias de datos se interpretan de for-

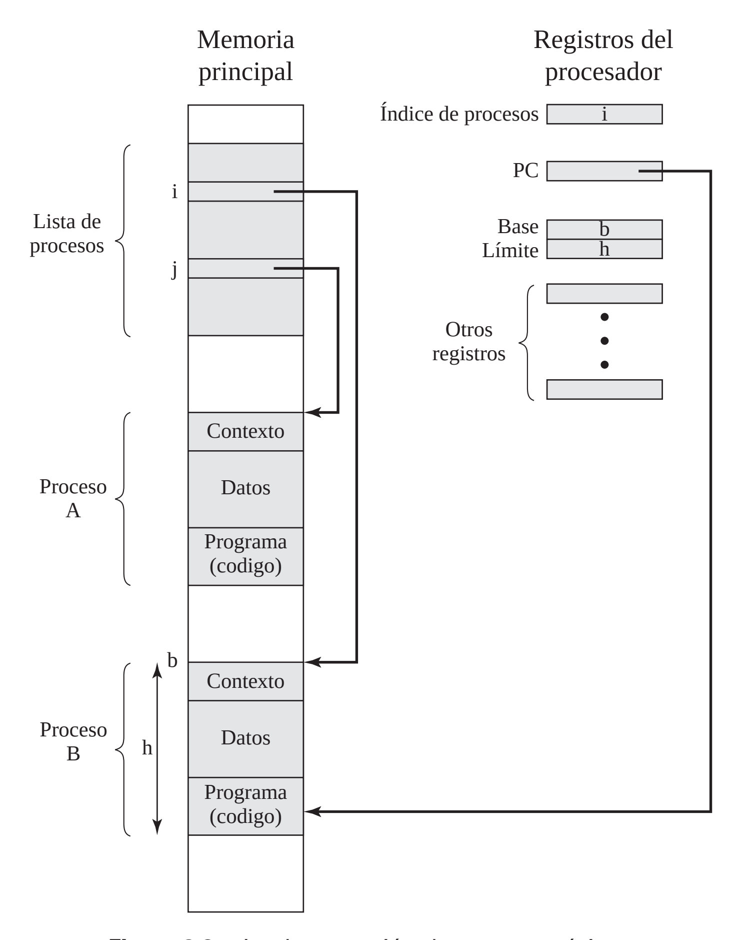


Figura 2.8. Implementación de procesos típica.

ma relativa al registro base y no deben exceder el valor almacenado en el registro límite. Esto previene la interferencia entre los procesos.

En la Figura 2.8, el registro índice del proceso indica que el proceso B está ejecutando. El proceso A estaba ejecutando previamente, pero fue interrumpido temporalmente. Los contenidos de todos los registros en el momento de la interrupción de A fueron guardados en su contexto de ejecución. Posteriormente, el sistema operativo puede cambiar el proceso en ejecución y continuar la ejecución del contexto de A. Cuando se carga el contador de programa con un valor que apunta al área de programa de A, el proceso A continuará la ejecución automáticamente.

Por tanto, el proceso puede verse como una estructura de datos. Un proceso puede estar en ejecución o esperando ejecutarse. El **estado** completo del proceso en un instante dado se contiene en su contexto. Esta estructura permite el desarrollo de técnicas potentes que aseguran la coordinación y la cooperación entre los procesos. Se pueden diseñar e incorporar nuevas características en el sistema operativo (por ejemplo, la prioridad), expandiendo el contexto para incluir cualquier información nueva que se utilice para dar soporte a dicha característica. A lo largo del libro, veremos un gran número de ejemplos donde se utiliza esta estructura de proceso para resolver los problemas provocados por la multiprogramación o la compartición de recursos.

GESTIÓN DE MEMORIA

Un entorno de computación que permita programación modular y el uso flexible de los datos puede ayudar a resolver mejor las necesidades de los usuarios. Los gestores de sistema necesitan un control eficiente y ordenado de la asignación de los recursos. Para satisfacer estos requisitos, el sistema operativo tiene cinco responsabilidades principales de gestión de almacenamiento:

- **Aislamiento de procesos.** El sistema operativo debe evitar que los procesos independientes interfieran en la memoria de otro proceso, tanto datos como instrucciones.
- **Asignación y gestión automática.** Los programas deben tener una asignación dinámica de memoria por demanda, en cualquier nivel de la jerarquía de memoria. La asignación debe ser transparente al programador. Por tanto, el programador no debe preocuparse de aspectos relacionados con limitaciones de memoria, y el sistema operativo puede lograr incrementar la eficiencia, asignando memoria a los trabajos sólo cuando se necesiten.
- **Soporte a la programación modular.** Los programadores deben ser capaces de definir módulos de programación y crear, destruir, y alterar el tamaño de los módulos dinámicamente.
- **Protección y control de acceso.** La compartición de memoria, en cualquier nivel de la jerarquía de memoria, permite que un programa direccione un espacio de memoria de otro proceso. Esto es deseable cuando se necesita la compartición por parte de determinadas aplicaciones. Otras veces, esta característica amenaza la integridad de los programas e incluso del propio sistema operativo. El sistema operativo debe permitir que varios usuarios puedan acceder de distintas formas a porciones de memoria.
- **Almacenamiento a largo plazo.** Muchas aplicaciones requieren formas de almacenar la información durante largos periodos de tiempo, después de que el computador se haya apagado.

Normalmente, los sistemas operativos alcanzan estos requisitos a través del uso de la memoria virtual y las utilidades de los sistemas operativos. El sistema operativo implementa un almacenamiento a largo plazo, con la información almacenada en objetos denominados ficheros. El fichero es un concepto lógico, conveniente para el programador y es una unidad útil de control de acceso y protección para los sistemas operativos.

La memoria virtual es una utilidad que permite a los programas direccionar la memoria desde un punto de vista lógico, sin importar la cantidad de memoria principal física disponible. La memoria virtual fue concebida como un método para tener múltiples trabajos de usuario residiendo en memoria principal de forma concurrente, de forma que no exista un intervalo de tiempo de espera entre la ejecución de procesos sucesivos, es decir, mientras un proceso se escribe en almacenamiento secundario y se lee el proceso sucesor. Debido a que los procesos varían de tamaño, si el procesador planifica un determinado número de procesos, es difícil almacenarlos compactamente en memoria principal. Se introdujeron los sistemas de paginación, que permiten que los procesos se compriman en un número determinado de bloques de tamaño fijo, denominados páginas. Un programa referencia una palabra por medio de una **dirección virtual**, que consiste en un número de página y un desplazamiento dentro de la página. Cada página de un proceso se puede localizar en cualquier sitio de memoria principal. El sistema de paginación proporciona una proyección dinámica entre las direcciones virtuales utilizadas en el programa y una **dirección real**, o dirección física, de memoria principal.

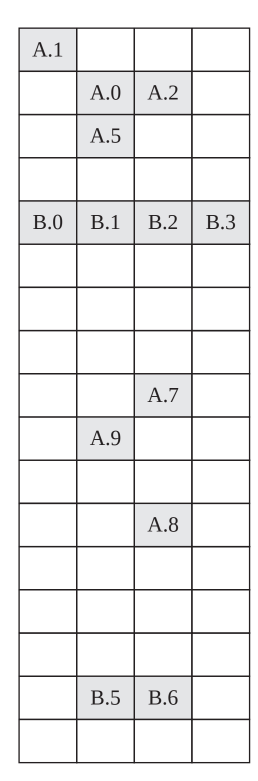
Con el hardware de proyección dinámica disponible, el siguiente paso era eliminar el requisito de que todas las páginas de un proceso residan en memoria principal simultáneamente. Todas las páginas de un proceso se mantienen en disco. Cuando un proceso está en ejecución, algunas de sus páginas se encuentran en memoria principal. Si se referencia una página que no está en memoria principal, el hardware de gestión de memoria lo detecta y permite que la página que falta se cargue. Dicho esquema se denomina área de memoria virtual y está representado en la Figura 2.9.

El hardware del procesador, junto con el sistema operativo, dota al usuario de un «procesador virtual» que tiene acceso a la memoria virtual. Este almacen podría ser un espacio de almacenamiento lineal o una colección de segmentos, que son bloques de longitud variable de direcciones contiguas. En ambos casos, las instrucciones del lenguaje de programación pueden referenciar al programa y a las ubicaciones de los datos en el área de memoria virtual. El aislamiento de los procesos se puede lograr dando a cada proceso una única área de memoria virtual, que no se solape con otras áreas. La compartición de memoria se puede lograr a través de porciones de dos espacios de memoria virtual que se solapan. Los ficheros se mantienen en un almacenamiento a largo plazo. Los ficheros o parte de los mismos se pueden copiar en la memoria virtual para que los programas los manipulen.

La Figura 2.10 destaca los aspectos de direccionamiento de un esquema de memoria virtual. El almacenamiento está compuesto por memoria principal directamente direccionable (a través de instrucciones máquina) y memoria auxiliar de baja velocidad a la que se accede de forma indirecta, cargando bloques de la misma en memoria principal. El hardware de traducción de direcciones (*memory management unit*: unidad de gestión de memoria) se interpone entre el procesador y la memoria. Los programas hacen referencia a direcciones virtuales, que son proyectadas sobre direcciones reales de memoria principal. Si una referencia a una dirección virtual no se encuentra en memoria física, entonces una porción de los contenidos de memoria real son llevados a la memoria auxiliar y los contenidos de la memoria real que se están buscando, son llevados a memoria principal. Durante esta tarea, el proceso que generó la dirección se suspende. El diseñador del sistema operativo necesita desarrollar un mecanismo de traducción de direcciones que genere poca sobrecarga y una política de asignación de almacenamiento que minimice el tráfico entre los diferentes niveles de la jerarquía de memoria.

PROTECCIÓN Y SEGURIDAD DE INFORMACIÓN

El crecimiento del uso de los sistemas de tiempo compartido y, más recientemente, las redes de computadores ha originado un incremento de la preocupación por la protección de la información. La naturaleza de las amenazas que conciernen a una organización variarán enormemente dependiendo de



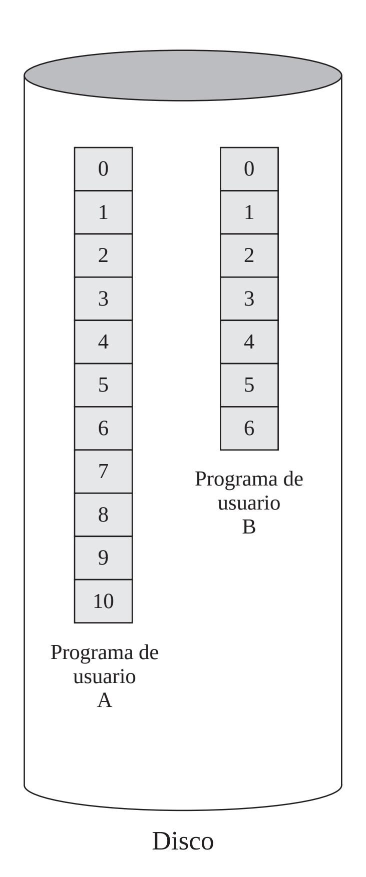


Figura 2.9. Conceptos de memoria virtual.

las circunstancias. Sin embargo, hay algunas herramientas de propósito general que se pueden utilizar en los computadores y sistemas operativos para soportar una gran variedad de mecanismos de protección y seguridad. En general, el principal problema es el control del acceso a los sistemas de computación y a la información almacenada en ellos.

La mayoría del trabajo en seguridad y protección relacionado con los sistemas operativos se puede agrupar de forma genérica en cuatro categorías:

- **Disponibilidad.** Relacionado con la protección del sistema frente a las interrupciones.
- **Confidencialidad.** Asegura que los usuarios no puedan leer los datos sobre los cuales no tienen autorización de acceso.
- **Integridad de los datos.** Protección de los datos frente a modificaciones no autorizadas.
- **Autenticidad.** Relacionado con la verificación apropiada de la identidad de los usuarios y la validez de los mensajes o los datos.

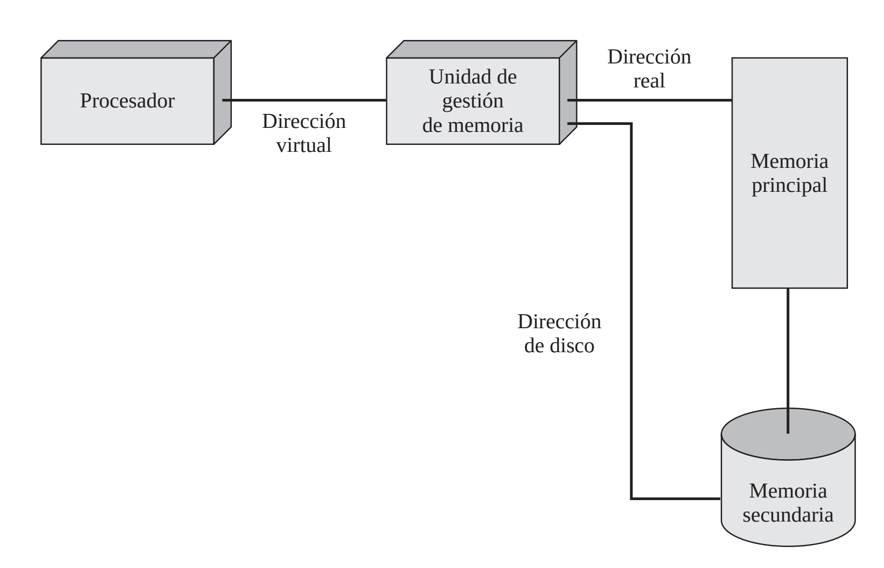


Figura 2.10. Direccionamiento de memoria virtual.

PLANIFICACIÓN Y GESTIÓN DE LOS RECURSOS

Una responsabilidad clave de los sistemas operativos es la gestión de varios recursos disponibles para ellos (espacio de memoria principal, dispositivos de E/S, procesadores) y para planificar su uso por parte de los distintos procesos activos. Cualquier asignación de recursos y política de planificación debe tener en cuenta tres factores:

- **Equitatividad.** Normalmente, se desea que todos los procesos que compiten por un determinado recurso, se les conceda un acceso equitativo a dicho recurso. Esto es especialmente cierto para trabajos de la misma categoría, es decir, trabajos con demandas similares.
- **Respuesta diferencial.** Por otro lado, el sistema operativo puede necesitar discriminar entre diferentes clases de trabajos con diferentes requisitos de servicio. El sistema operativo debe tomar las decisiones de asignación y planificación con el objetivo de satisfacer el conjunto total de requisitos. Además, debe tomar las decisiones de forma dinámica. Por ejemplo, si un proceso está esperando por el uso de un dispositivo de E/S, el sistema operativo puede intentar planificar este proceso para su ejecución tan pronto como sea posible a fin de liberar el dispositivo para posteriores demandas de otros procesos.
- **Eficiencia.** El sistema operativo debe intentar maximizar la productividad, minimizar el tiempo de respuesta, y, en caso de sistemas de tiempo compartido, acomodar tantos usuarios como sea posible. Estos criterios entran en conflicto; encontrar un compromiso adecuado en una situación particular es un problema objeto de la investigación sobre sistemas operativos.

La planificación y la gestión de recursos son esencialmente problemas de investigación, y se pueden aplicar los resultados matemáticos de esta disciplina. Adicionalmente, medir la actividad del sistema es importante para ser capaz de monitorizar el rendimiento y realizar los ajustes correspondientes.

La Figura 2.11 sugiere los principales elementos del sistema operativo relacionados con la planificación de procesos y la asignación de recursos en un entorno de multiprogramación. El sistema operativo mantiene un número de colas, cada una de las cuales es simplemente una lista de procesos esperando por algunos recursos. La cola a corto plazo está compuesta por procesos que se encuentran

en memoria principal (o al menos una porción mínima esencial de cada uno de ellos está en memoria principal) y están listos para ejecutar, siempre que el procesador esté disponible. Cualquiera de estos procesos podría usar el procesador a continuación. Es responsabilidad del planificador a corto plazo, o *dispatcher*, elegir uno de ellos. Una estrategia común es asignar en orden a cada proceso de la cola un intervalo de tiempo; esta técnica se conoce como ***round-robin*** o **turno rotatorio**. En efecto, la técnica de turno rotatorio emplea una cola circular. Otra estrategia consiste en asignar niveles de prioridad a los distintos procesos, siendo el planificador el encargado de elegir los procesos en orden de prioridad.

La cola a largo plazo es una lista de nuevos trabajos esperando a utilizar el procesador. El sistema operativo añade trabajos al sistema transfiriendo un proceso desde la cola a largo plazo hasta la cola a corto plazo. En este punto, se debe asignar una porción de memoria principal al proceso entrante. Por tanto, el sistema operativo debe estar seguro de que no sobrecarga la memoria o el tiempo de procesador admitiendo demasiados procesos en el sistema. Hay una cola de E/S por cada dispositivo de E/S. Más de un proceso puede solicitar el uso del mismo dispositivo de E/S. Todos los procesos que esperan utilizar dicho dispositivo, se encuentran alineados en la cola del dispositivo. De nuevo, el sistema operativo debe determinar a qué proceso le asigna un dispositivo de E/S disponible.

Si ocurre una interrupción, el sistema operativo recibe el control del procesador a través de un manejador de interrupciones. Un proceso puede invocar específicamente un servicio del sistema operativo, tal como un manejador de un dispositivo de E/S, mediante una llamada a sistema. En este caso, un manejador de la llamada a sistema es el punto de entrada al sistema operativo. En cualquier caso, una vez que se maneja la interrupción o la llamada a sistema, se invoca al planificador a corto plazo para que seleccione un proceso para su ejecución.

Lo que se ha detallado hasta ahora es una descripción funcional; los detalles y el diseño modular de esta porción del sistema operativo difiere en los diferentes sistemas. Gran parte del esfuerzo en la investigación y desarrollo de los sistemas operativos ha sido dirigido a la creación de algoritmos de planificación y estructuras de datos que proporcionen equitatividad, respuesta diferencial y eficiencia.

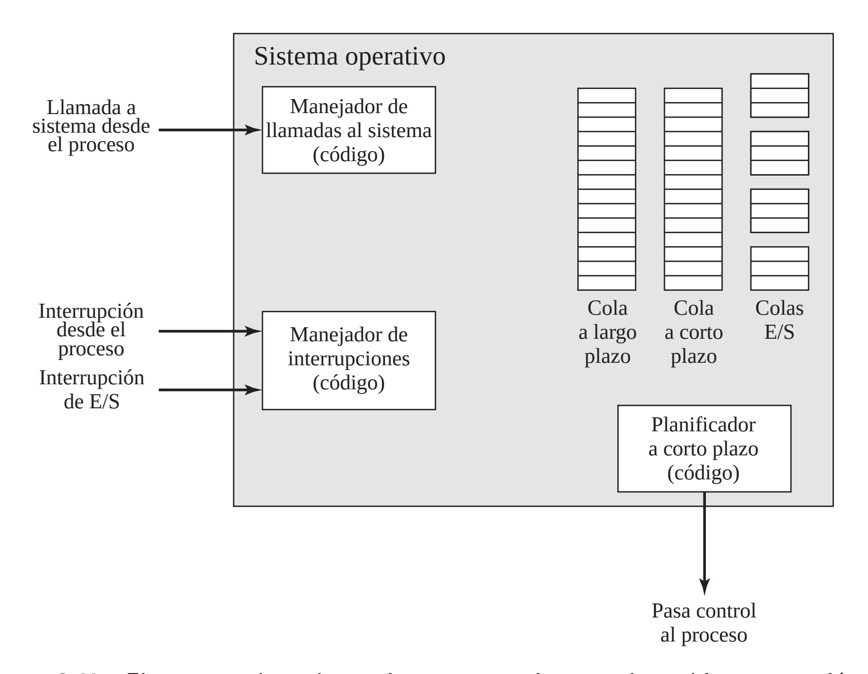


Figura 2.11. Elementos clave de un sistema operativo para la multiprogramación.

ESTRUCTURA DEL SISTEMA

A medida que se han añadido más características a los sistemas operativos y el hardware subyacente se ha vuelto más potente y versátil, han crecido el tamaño y la complejidad de los sistemas operativos. CTSS, que fue puesto en marcha en el MIT en 1963, estaba compuesto aproximadamente por 32.000 palabras de almacenamiento de 36 bits. OS/360, presentado por IBM un año después, tenía más de un millón de instrucciones de máquina. En 1975, el sistema Multics, desarrollado por MIT y los laboratorios Bell, había superado los 20 millones de instrucciones. Es cierto que más recientemente, se han introducido algunos sistemas operativos más sencillos para sistemas más pequeños, pero éstos inevitablemente se hacen más complejos a medida que el hardware subyacente y los requisitos de usuario se incrementan. Por tanto, el sistema UNIX de hoy es muchísimo más complejo que el sistema casi de juguete puesto en marcha por unos pocos programadores de gran talento a comienzo de los años 70, y el sencillo sistema MS-DOS supuso el comienzo de los ricos y complejos sistemas OS/2 y Windows. Por ejemplo, Windows NT 4.0 contiene 16 millones de líneas de código y Windows 2000 duplica este número.

El tamaño de un sistema operativo con un conjunto completo de características, y la dificultad del problema que afronta dicho sistema, ha llevado a esta disciplina a cuatro desafortunados, aunque demasiado comunes, problemas. En primer lugar, los sistemas operativos se entregan tarde de forma crónica. Esto implica la creación de nuevos sistemas operativos y frecuentes actualizaciones a viejos sistemas. En segundo lugar, los sistemas tienen fallos latentes que deben ser planteados y resueltos. En tercer lugar, el rendimiento no es frecuentemente el esperado. En último lugar, se ha comprobado que es imposible construir un sistema operativo complejo que no sea vulnerable a una gran cantidad de ataques de seguridad, incluyendo virus, gusanos y accesos no autorizados.

Para gestionar la complejidad de los sistemas operativos y eliminar estos problemas, se ha puesto mucho énfasis en la estructura software del sistema operativo a lo largo de los años. Ciertos puntos parecen obvios. El software debe ser modular. Esto ayudará a organizar el proceso de desarrollo de software y limitará el esfuerzo de diagnosticar y corregir errores. Los módulos deben tener interfaces bien definidas, y estas interfaces deben ser tan sencillas como sea posible. De nuevo, esto facilita la programación. También facilita la evolución del sistema. Con mínimas interfaces entre los módulos, se puede modificar un módulo con un mínimo impacto en otros módulos.

Para sistemas operativos grandes, que ejecutan desde millones a decenas de millones de líneas de código, no es suficiente la programación modular. De hecho, ha habido un incremento en el uso de los conceptos de capas jerárquicas y abstracción de información. La estructura jerárquica de un sistema operativo moderno separa sus funciones de acuerdo a las características de su escala de tiempo y su nivel de abstracción. Se puede ver el sistema como una serie de niveles. Cada nivel realiza un subconjunto relacionado de funciones requeridas por el sistema operativo. Dicho nivel confía en los niveles inmediatamente inferiores para realizar funciones más primitivas y ocultar los detalles de esas funciones. Cada nivel proporciona servicios a la capa inmediatamente superior. Idealmente, estos niveles deben definirse de tal forma que los cambios en un nivel no requieran cambios en otros niveles. Por tanto, de esta forma se ha descompuesto un problema en un número de subproblemas más manejables.

En general, las capas inferiores tratan con una escala de tiempo menor. Algunas partes del sistema operativo deben interaccionar directamente con el hardware del computador, donde los eventos pueden tener una escala de tiempo tan ínfima como unas pocas mil millonésimas de segundo. En el otro extremo del espectro, algunas partes del sistema operativo se comunican con el usuario, que invoca mandatos en una escala de tiempo mucho más larga, tal vez unos pocos segundos. El uso de un conjunto de niveles se adapta adecuadamente a este entorno.

La forma en que estos principios se aplican varía enormemente entre los sistemas operativos contemporáneos. Sin embargo, es útil en este punto, para el propósito de mostrar los sistemas operativos,

presentar un modelo de sistema operativo jerárquico. Uno propuesto en [BROW84] y [DENN84] es útil, aunque no corresponde a ningún sistema operativo particular. El modelo se define en la Tabla 2.4 y está compuesto por los siguientes niveles:

- **Nivel 1.** Está formado por circuitos electrónicos, donde los objetos tratados son registros, celdas de memoria, y puertas lógicas. Las operaciones definidas en estos objetos son acciones, como poner a cero un registro o leer una posición de memoria.
- **Nivel 2.** El conjunto de instrucciones del procesador. Las operaciones a este nivel son aquéllas permitidas en el conjunto de instrucciones de lenguaje máquina, como adición, resta, carga o almacenamiento.
- **Nivel 3.** Añade el concepto de procedimiento o subrutina, más las operaciones de llamada y retorno (*call/return*).
- **Nivel 4.** Introduce las interrupciones, que permiten al procesador guardar el contexto actual e invocar una rutina de tratamiento de interrupciones.

Tabla 2.4. Jerarquía de diseño del sistema operativo.

Nivel	Nombre	Objetos	Ejemplos de operaciones
13	Intérprete de mandatos	Entorno de programación de usuario	Sentencias en lenguaje del intérprete de mandatos
12	Procesos de usuario	Procesos de usuario	Salir, matar, suspender, continuar
11	Directorios	Directorios	Crear, destruir, insertar entrada, eliminar entrada, buscar, listar
10	Dispositivos	Dispositivos externos, como impresoras, pantallas y teclados	Abrir, cerrar, leer, escribir
9	Sistema de ficheros	Ficheros	Crear, destruir, abrir, cerrar, leer, escribir
8	Comunicaciones	Tuberías	Crear, destruir, abrir, cerrar, leer, escribir
7	Memoria virtual	Segmentos, páginas	Leer, escribir, cargar
6	Almacenamiento secundario local	Bloques de datos, canales de dispositivo	Leer, escribir, asignar, liberar
5	Procesos primitivos	Procesos primitivos, semáforos, lista de procesos listos	Suspender, continuar, esperar, señalizar
4	Interrupciones	Programas de gestión de interrupciones	Invocar, enmascarar, desenmascarar, reintentar
3	Procedimientos	Procedimientos, pila de llamadas, registro de activación	Marcar la pila, llamar, retornar
2	Conjunto de instrucciones	Pila de evaluación, intérprete de microprogramas, datos escalares y vectoriales	Cargar, almacenar, sumar, restar, saltar
1	Circuitos electrónicos	Registros, puertas, buses, etc.	Poner a 0, transferir, activar, complementar

El área sombreado en gris representa al hardware.

Estos cuatros primeros niveles no son parte del sistema operativo, sino que constituyen el hardware del procesador. Sin embargo, algunos elementos del sistema operativo se empiezan a mostrar en estos niveles, por ejemplo, las rutinas de tratamiento de interrupciones. Es el Nivel 5 el que corresponde con el sistema operativo propiamente dicho y en el que los conceptos asociados a la multiprogramación aparecen.

- **Nivel 5.** En este nivel se introduce la noción de un proceso como un programa en ejecución. Los requisitos fundamentales de los sistemas operativos para dar soporte a múltiples procesos incluyen la habilidad de suspender y continuar los procesos. Esto requiere guardar los registros de hardware de forma que se pueda interrumpir la ejecución de un proceso e iniciar la de otro. Adicionalmente, si los procesos necesitan cooperar, se necesitan algunos métodos de sincronización. Una de las técnicas más sencillas, y un concepto importante en el diseño de los sistemas operativos, es el semáforo, una técnica de señalización sencilla que se analiza en el Capítulo 5.
- **Nivel 6.** Trata los dispositivos de almacenamiento secundario del computador. En este nivel, se dan la funciones para posicionar las cabezas de lectura/escritura y la transferencia real de bloques. El Nivel 6 delega al Nivel 5 la planificación de la operación y la notificación al proceso solicitante de la finalización de la misma. Niveles más altos se preocupan de la dirección en el disco de los datos requeridos y proporcionan una petición del bloque de datos apropiado a un controlador de dispositivo del Nivel 5.
- **Nivel 7.** Crea un espacio de direcciones lógicas para los procesos. Este nivel organiza el espacio de direcciones virtuales en bloques que pueden moverse entre memoria principal y memoria secundaria. Tres esquemas son de uso común: aquéllos que utilizan páginas de tamaño fijo, aquéllos que utilizan segmentos de longitud variable y aquéllos que utilizan ambos. Cuando un bloque de memoria necesario no se encuentra en memoria principal, la lógica de este nivel requiere una transferencia desde el Nivel 6.

Hasta este punto, el sistema operativo trata con los recursos de un único procesador. Comenzando con el Nivel 8, el sistema operativo trata con objetos externos, como dispositivos periféricos y posiblemente redes y computadores conectados a la red. Los objetos de estos niveles superiores son objetos lógicos con nombre, que pueden compartirse entre procesos del mismo computador o entre múltiples computadores.

- **Nivel 8.** Trata con la comunicación de información y mensajes entre los procesos. Mientras que el Nivel 5 proporciona un mecanismo de señalización primitivo que permite la sincronización de procesos, este nivel trata con una compartición de información más rica. Una de las herramientas más potentes para este propósito es la tubería o *pipe*, que es un canal lógico para el flujo de datos entre los procesos. Una tubería se define por su salida de un proceso y su entrada en otro proceso. Se puede también utilizar para enlazar dispositivos externos o ficheros a procesos. El concepto se discute en el Capítulo 6.
- **Nivel 9.** Da soporte al almacenamiento a largo plazo en ficheros con nombre. En este nivel, los datos en el almacenamiento secundario se ven en términos de entidades abstractas y con longitud variable. Esto contrasta con la visión orientada a hardware del almacenamiento secundario en términos de pistas, sectores y bloques de tamaño fijo en el Nivel 6.
- **Nivel 10.** Proporciona acceso a los dispositivos externos utilizando interfaces estándar.
- **Nivel 11.** Es el nivel responsable para mantener la asociación entre los identificadores externos e internos de los recursos y objetos del sistema. El identificador externo es un nombre que puede utilizar una aplicación o usuario. El identificador interno es una dirección de otro identificador que puede utilizarse por parte de los niveles inferiores del sistema operativo para lo-

calizar y controlar un objeto. Estas asociaciones se mantienen en un directorio. Las entradas no sólo incluyen la asociación entre identificadores externos/internos, sino también características como los derechos de acceso.

- **Nivel 12.** Proporciona una utilidad completa para dar soporte a los procesos. Este nivel va más allá del proporcionado por el Nivel 5. En el Nivel 5, sólo se mantienen los contenidos de los registros del procesador asociados con un proceso, más la lógica de los procesos de planificación. En el Nivel 12, se da soporte a toda la información necesaria para la gestión ordenada de los procesos. Esto incluye el espacio de direcciones virtuales de los procesos, una lista de objetos y procesos con la cual puede interactuar y las restricciones de esta interacción, parámetros pasados al proceso en la creación. También se incluye cualquier otra característica que pudiera utilizar el sistema operativo para controlar el proceso.
- **Nivel 13.** Proporciona una interfaz del sistema operativo al usuario. Se denomina ***shell*** (caparazón), porque separa al usuario de los detalles de los sistemas operativos y presenta el sistema operativo simplemente como una colección de servicios. El *shell* acepta mandatos de usuario o sentencias de control de trabajos, los interpreta y crea y controla los procesos que necesita para su ejecución. Por ejemplo, a este nivel la interfaz podría implementarse de una manera gráfica, proporcionando al usuario mandatos a través de una lista presentada como un menú y mostrando los resultados utilizando una salida gráfica conectada a un dispositivo específico, tal y como una pantalla.

Este modelo hipotético de un sistema operativo proporciona una estructura descriptiva útil y sirve como una guía de implementación. El lector puede volver a estudiar esta estructura durante el desarrollo del libro, para situar cualquier aspecto particular de diseño bajo discusión en el contexto apropiado.

2.4. DESARROLLOS QUE HAN LLEVADO A LOS SISTEMAS OPERATIVOS MODERNOS

A lo largo de los años, ha habido una evolución gradual de la estructura y las capacidades de los sistemas operativos. Sin embargo, en los últimos años se han introducido un gran número de nuevos elementos de diseño tanto en sistemas operativos nuevos como en nuevas versiones de sistemas operativos existentes que han creado un cambio fundamental en la naturaleza de los sistemas operativos. Estos sistemas operativos modernos responden a nuevos desarrollos en hardware, nuevas aplicaciones y nuevas amenazas de seguridad. Entre las causas de hardware principales se encuentran las máquinas multiprocesador, que han logrado incrementar la velocidad de la máquina en mayor medida, los dispositivos de conexión de alta velocidad a la red, y el tamaño creciente y variedad de dispositivos de almacenamiento de memoria. En el campo de las aplicaciones, las aplicaciones multimedia, Internet y el acceso a la Web, y la computación cliente/servidor han influido en el diseño del sistema operativo. Respecto a la seguridad, el acceso a Internet de los computadores ha incrementado en gran medida la amenaza potencial y ataques sofisticados, tales como virus, gusanos, y técnicas de *hacking*, lo que ha supuesto un impacto profundo en el diseño de los sistemas operativos.

La velocidad de cambio en las demandas de los sistemas operativos requiere no sólo modificaciones o mejoras en arquitecturas existentes sino también nuevas formas de organizar el sistema operativo. Un amplio rango de diferentes técnicas y elementos de diseño se han probado tanto en sistemas operativos experimentales como comerciales, pero la mayoría de este trabajo encaja en las siguientes categorías:

- Arquitectura micronúcleo o *microkernel*.
- Multihilo.
- Multiprocesamiento simétrico.

- Sistemas operativos distribuidos.
- Diseño orientado a objetos.

Hasta hace relativamente poco tiempo, la mayoría de los sistemas operativos estaban formados por un gran **núcleo monolítico**. Estos grandes núcleos proporcionan la mayoría de las funcionalidades consideradas propias del sistema operativo, incluyendo la planificación, los sistemas de ficheros, las redes, los controladores de dispositivos, la gestión de memoria y otras funciones. Normalmente, un núcleo monolítico se implementa como un único proceso, con todos los elementos compartiendo el mismo espacio de direcciones. Una **arquitectura micronúcleo** asigna sólo unas pocas funciones esenciales al núcleo, incluyendo los espacios de almacenamiento, comunicación entre procesos (IPC), y la planificación básica. Ciertos procesos proporcionan otros servicios del sistema operativo, algunas veces denominados servidores, que ejecutan en modo usuario y son tratados como cualquier otra aplicación por el micronúcleo. Esta técnica desacopla el núcleo y el desarrollo del servidor. Los servidores pueden configurarse para aplicaciones específicas o para determinados requisitos del entorno. La técnica micronúcleo simplifica la implementación, proporciona flexibilidad y se adapta perfectamente a un entorno distribuido. En esencia, un micronúcleo interactúa con procesos locales y remotos del servidor de la misma forma, facilitando la construcción de los sistemas distribuidos.

Multitheading es una técnica en la cual un proceso, ejecutando una aplicación, se divide en una serie de hilos o *threads* que pueden ejecutar concurrentemente. Se pueden hacer las siguientes distinciones:

- ***Thread* o hilo.** Se trata de una unidad de trabajo. Incluye el contexto del procesador (que contiene el contador del programa y el puntero de pila) y su propia área de datos para una pila (para posibilitar el salto a subrutinas). Un hilo se ejecuta secuencialmente y se puede interrumpir de forma que el procesador pueda dar paso a otro hilo.
- **Proceso.** Es una colección de uno o más hilos y sus recursos de sistema asociados (como la memoria, conteniendo tanto código, como datos, ficheros abiertos y dispositivos). Esto corresponde al concepto de programa en ejecución. Dividiendo una sola aplicación en múltiples hilos, el programador tiene gran control sobre la modularidad de las aplicaciones y la temporización de los eventos relacionados con la aplicación.

La técnica *multithreading* es útil para las aplicaciones que llevan a cabo un número de tareas esencialmente independientes que no necesitan ser serializadas. Un ejemplo es un servidor de bases de datos que escucha y procesa numerosas peticiones de cliente. Con múltiples hilos ejecutándose dentro del mismo proceso, intercambiar la ejecución entre los hilos supone menos sobrecarga del procesador que intercambiar la ejecución entre diferentes procesos pesados. Los hilos son también útiles para estructurar procesos que son parte del núcleo del sistema operativo, como se describe en los capítulos siguientes.

Hasta hace poco tiempo, los computadores personales y estaciones de trabajo virtualmente de un único usuario contenían un único procesador de propósito general. A medida que la demanda de rendimiento se incrementa y el coste de los microprocesadores continúa cayendo, los fabricantes han introducido en el mercado computadores con múltiples procesadores. Para lograr mayor eficiencia y fiabilidad, una técnica consiste en emplear **multiprocesamiento simétrico (SMP: *Symmetric Multi-Processing*)**, un término que se refiere a la arquitectura hardware del computador y también al comportamiento del sistema operativo que explota dicha arquitectura. Se puede definir un multiprocesador simétrico como un sistema de computación aislado con las siguientes características:

1. Tiene múltiples procesadores.
2. Estos procesadores comparten las mismas utilidades de memoria principal y de E/S, interconectadas por un bus de comunicación u otro esquema de conexión interna.
3. Todos los procesadores pueden realizar las mismas funciones (de ahí el término *simétrico*).

El sistema operativo de un SMP planifica procesos o hilos a través de todos los procesadores. SMP tiene diversas ventajas potenciales sobre las arquitecturas monoprocesador, entre las que se incluyen:

- **Rendimiento.** Si el trabajo se puede organizar de tal forma que alguna porción del trabajo se pueda realizar en paralelo, entonces un sistema con múltiples procesadores alcanzará mayor rendimiento que uno con un solo procesador del mismo tipo. Esto se muestra en la Figura 2.12. Con la multiprogramación, sólo un proceso puede ejecutar a la vez; mientras tanto, el resto de los procesos esperan por el procesador. Con multiproceso, más de un proceso puede ejecutarse simultáneamente, cada uno de ellos en un procesador diferente.
- **Disponibilidad.** En un multiprocesador simétrico, debido a que todos los procesadores pueden llevar a cabo las mismas funciones, el fallo de un solo procesador no para la máquina. Por el contrario, el sistema puede continuar funcionando con un rendimiento reducido.
- **Crecimiento incremental.** Un usuario puede mejorar el rendimiento de un sistema añadiendo un procesador adicional.
- **Escalado.** Los fabricantes pueden ofrecer un rango de productos con diferente precio y características basadas en el número de procesadores configurado en el sistema.

Es importante notar que estas características son beneficios potenciales, no garantizados. El sistema operativo debe proporcionar herramientas y funciones para explotar el paralelismo en un sistema SMP.

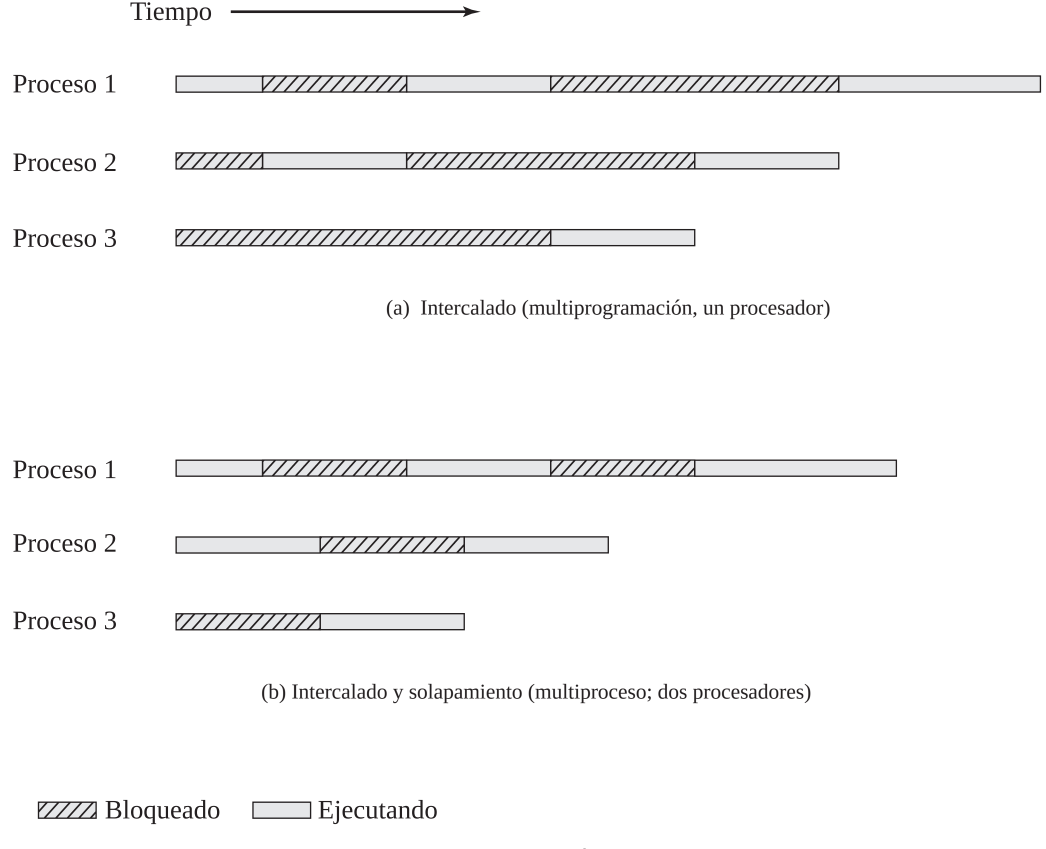


Figura 2.12. Multiprogramación y multiproceso.

La técnica *multithreading* y SMP son frecuentemente analizados juntos, pero son dos utilidades independientes. Incluso en un nodo monoprocesador, la técnica de *multihreading* es útil para estructurar aplicaciones y procesos de núcleo. Una máquina SMP es útil incluso para procesos que no contienen hilos, porque varios procesos pueden ejecutar en paralelo. Sin embargo, ambas utilidades se complementan y se pueden utilizar de forma conjunta efectivamente.

Una característica atractiva de un SMP es que la existencia de múltiples procesadores es transparente al usuario. El sistema operativo se encarga de planificar los hilos o procesos en procesadores individuales y de la sincronización entre los procesadores. Este libro discute la planificación y los mecanismos de sincronización utilizados para proporcionar una apariencia de único sistema al usuario. Un problema diferente es proporcionar la apariencia de un solo sistema a un cluster de computadores separado-un sistema multicomputador. En este caso, se trata de una colección de entidades (computadores) cada uno con sus propios módulos de memoria principal, de memoria secundaria y otros módulos de E/S.

Un **sistema operativo distribuido** proporciona la ilusión de un solo espacio de memoria principal y un solo espacio de memoria secundario, más otras utilidades de acceso unificadas, como un sistema de ficheros distribuido. Aunque los clusters se están volviendo cada día más populares, y hay muchos productos para clusters en el mercado, el estado del arte de los sistemas distribuidos está retrasado con respecto a los monoprocesadores y sistemas operativos SMP. Se examinarán dichos sistemas en la Parte 6.

Otra innovación en el diseño de los sistemas operativos es el uso de tecnologías orientadas a objetos. El **diseño orientado a objetos** introduce una disciplina al proceso de añadir extensiones modulares a un pequeño núcleo. A nivel del sistema operativo, una estructura basada en objetos permite a los programadores personalizar un sistema operativo sin eliminar la integridad del sistema. La orientación a objetos también facilita el desarrollo de herramientas distribuidas y sistemas operativos distribuidos.

2.5. DESCRIPCIÓN GLOBAL DE MICROSOFT WINDOWS

En esta sección se va a realizar una descripción global de Microsoft Windows; se describirá UNIX en la siguiente sección.

HISTORIA

La historia de Windows comienza con un sistema operativo muy diferente, desarrollado por Microsoft para el primer computador personal IBM y conocido como MS-DOS o PC-DOS. La versión inicial, DOS 1.0 apareció en 1981. Estaba compuesto por 4000 líneas de código fuente en ensamblador y ejecutaba en 8 Kbytes de memoria, utilizando el microprocesador Intel 8086.

Cuando IBM desarrolló un computador personal basado en disco duro, el PC XT, Microsoft desarrolló DOS 2.0, que salió al mercado en 1983. Este sistema daba soporte al disco duro y proporcionaba jerarquía de directorios. Hasta ese momento, un disco podía contener sólo un directorio con ficheros, soportando un máximo de 64 ficheros. Mientras que esto era adecuado en la era de los disquetes, era demasiado limitado para los discos duros, y la restricción de un solo directorio era demasiado burda. Esta nueva versión permitía que los directorios contuvieran tantos subdirectorios como ficheros. La nueva versión también contenía un conjunto de mandatos más rico dentro del sistema operativo, que proporcionaban funciones que eran realizadas como programas externos con la versión 1. Entre las capacidades añadidas se encontraban algunas características de los sistemas UNIX, como la redirección de E/S, que consiste en la capacidad para modificar la entrada o salida de una determinada aplicación, y la impresión en segundo plano. La porción de memoria residente creció a 24 Kbytes.

Cuando IBM anunció el PC AT en 1984, Microsoft introdujo DOS 3.0. El sistema AT contenía el procesador Intel 80286, que proporcionaba características de direccionamiento extendido y protección de memoria. Estas no eran utilizadas por DOS. Para que fuera compatible con versiones anteriores, el sistema operativo simplemente utilizaba el 80286 como un «8086 rápido». El sistema operativo sí daba soporte a un nuevo teclado y periféricos de disco duro. Incluso así, los requisitos de memoria se incrementaron a 36 Kbytes. Hubo varias actualizaciones notables de la versión 3.0. DOS 3.1, que apareció en 1984, daba soporte a la conexión a través de la red para PC. El tamaño de la porción residente no cambió; esto se logró incrementando la cantidad de sistema operativo que podía ser intercambiado (*swapped*). DOS 3.3, que apareció en 1987, daba soporte a la nueva línea de máquinas IBM, las PS/2. De nuevo, esta versión no se beneficiaba de las capacidades del procesador del PS/2, proporcionadas por el 80286 y los *chips* de 32 bits 80386. En este punto, la porción residente había alcanzado un mínimo de 46 Kbytes, incrementándose esta cantidad si se seleccionaban ciertas extensiones opcionales.

En este momento, DOS estaba utilizando el entorno muy por debajo de sus posibilidades. La introducción del 80486 y el *chip* Intel Pentium proporcionaban características que simplemente no podía explotar un sencillo sistema DOS. Mientras tanto, al comienzo de los años 80, Microsoft comenzó a desarrollar una interfaz gráfica de usuario (GUI: *Graphical User Interface*) que sería interpuesta entre el usuario y el sistema operativo DOS. El objetivo de Microsoft era competir con Macintosh, cuyo sistema operativo era insuperable por facilidad de uso. En 1990, Microsoft tenía una versión de la GUI, conocida como Windows 3.0, que incorporaba algunas de las características más amigables de Macintosh. Sin embargo, estaba todavía limitada por la necesidad de ejecutar encima de DOS.

Microsoft intentó el desarrollo de un sistema operativo de nueva generación con IBM para explotar la potencia de los nuevos microprocesadores, el cual incorporaría las características de facilidad de uso de Windows, pero este proyecto fue finalmente abortado. Después de este intento fallido, Microsoft desarrolló un nuevo y propio sistema operativo desde cero, que denominó Windows NT. Windows NT explota las capacidades de los microprocesadores contemporáneos y proporciona multitarea en un entorno mono o multiusuario.

La primera versión de Windows NT (3.1) apareció en 1993, con la misma interfaz gráfica que Windows 3.1, otro sistema operativo de Microsoft (el sucesor de Windows 3.0). Sin embargo, NT 3.1 era un nuevo sistema operativo de 32 bits con la capacidad de dar soporte a las aplicaciones Windows, a las antiguas aplicaciones de DOS y a OS/2.

Después de varias versiones de NT 3.x, Microsoft produjo NT 4.0. NT 4.0 tiene esencialmente la misma arquitectura interna que 3.x. El cambio externo más notable es que NT 4.0 proporciona la misma interfaz de usuario que Windows 95. El cambio arquitectónico más importante es que varios componentes gráficos que ejecutaban en modo usuario como parte del subsistema Win32 en 3.x se mueven al sistema ejecutivo de Windows NT, que ejecuta en modo núcleo. El beneficio de este cambio consiste en la aceleración de estas funciones importantes. La desventaja potencial es que estas funciones gráficas ahora tienen acceso a servicios de bajo nivel del sistema, que pueden impactar en la fiabilidad del sistema operativo.

En 2000, Microsoft introdujo la siguiente principal actualización, que se materializó en el sistema Windows 2000. De nuevo, el sistema ejecutivo subyacente y la arquitectura del núcleo son fundamentalmente los mismos que NT 4.0, pero se han añadido nuevas características. El énfasis en Windows 2000 es la adición de servicios y funciones que dan soporte al procesamiento distribuido. El elemento central de las nuevas características de Windows 2000 es *Active Directory*, que es un servicio de directorios distribuido capaz de realizar una proyección entre nombres de objetos arbitrarios y cualquier información sobre dichos objetos.

Un punto final general sobre Windows 2000 es la distinción entre Windows 2000 Server y el escritorio de Windows 2000. En esencia, el núcleo, la arquitectura ejecutiva y los servicios son los mismos, pero Windows 2000 Server incluye algunos servicios requeridos para su uso como servidor de red.

En 2001, apareció la última versión de escritorio de Windows, conocida como Windows XP. Se ofrecieron versiones de XP tanto de PC de hogar como de estación de trabajo de negocio. También en 2001, apareció una versión de XP de 64 bits. En el año 2003, Microsoft presentó una nueva versión de servidor, conocido como Windows Server 2003. Existen versiones disponibles de 32 y 64 bits. Las versiones de 64 bits de XP y Server 2003 están diseñadas específicamente para el hardware del Intel Itanium de 64 bits.

MULTITAREA MONOUSUARIO

Windows (desde Windows 2000 en adelante) es un ejemplo significativo de lo que significa la nueva ola en los sistemas operativos de microcomputadores (otros ejemplos son OS/2 y MacOS). El desarrollo de Windows fue dirigido por la necesidad de explotar las capacidades de los microprocesadores actuales de 32 bits, que rivalizan con los *mainframes* y minicomputadores de hace unos poco años en velocidad, sofisticaciones de hardware y capacidad de memoria.

Una de las características más significativas de estos nuevos sistemas operativos es que, aunque están todavía pensados para dar soporte a un único usuario interactivo, se trata de sistemas operativos multitarea. Dos principales desarrollos han disparado la necesidad de multitarea en computadores personales, estaciones de trabajo y servidores. En primer lugar, con el incremento de la velocidad y la capacidad de memoria de los microprocesadores, junto al soporte para memoria virtual, las aplicaciones se han vuelto más complejas e interrelacionadas. Por ejemplo, un usuario podría desear utilizar un procesador de texto, un programa de dibujo, y una hoja de cálculo simultáneamente para producir un documento. Sin multitarea, si un usuario desea crear un dibujo y copiarlo en un documento, se requieren los siguientes pasos:

1. Abrir el programa de dibujo.
2. Crear el dibujo y guardarlo en un fichero o en el portapapeles.
3. Cerrar el programa de dibujo.
4. Abrir el procesador de texto.
5. Insertar el dibujo en el sitio adecuado.

Si se desea cualquier cambio, el usuario debe cerrar el procesador de texto, abrir el programa de dibujo, editar la imagen gráfica, guardarlo, cerrar el programa de dibujo, abrir el procesador de texto e insertar la imagen actualizada. Este proceso se vuelve tedioso muy rápidamente. A medida que los servicios y capacidades disponibles para los usuarios se vuelven más potentes y variados, el entorno monotarea se vuelve más burdo y menos amigable. En un entorno multitarea, el usuario abre cada aplicación cuando la necesita, y la deja abierta. La información se puede mover entre las aplicaciones fácilmente. Cada aplicación tiene una o más ventanas abiertas, y una interfaz gráfica con un dispositivo puntero tal como un ratón permite al usuario navegar fácilmente en este entorno.

Una segunda motivación para la multitarea es el aumento de la computación cliente/servidor. En este paradigma, un computador personal o estación de trabajo (cliente) y un sistema *host* (servidor) se utilizan de forma conjunta para llevar a cabo una aplicación particular. Los dos están enlazados, y a cada uno se le asigna aquella parte del trabajo que se adapta a sus capacidades. El paradigma cliente/servidor se puede llevar a cabo en una red de área local formada por computadores personales y servidores o por medio de un enlace entre un sistema usuario y un gran *host*, como por ejemplo un *mainframe*. Una aplicación puede implicar a uno o más computadores personales y uno o más dispositivos de servidor. Para proporcionar la respuesta requerida, el sistema operativo necesita dar soporte al hardware de comunicación en tiempo real, los protocolos de comunicación asociados y las arquitecturas de transferencia de datos, y a la vez dar soporte a la interacción de los usuarios.

Las características que se describen a continuación se refieren a la versión Profesional de Windows. La versión Server (Servidor) es también multitarea pero debe permitir el uso de múltiples usuarios. Da soporte a múltiples conexiones locales de servidor y proporciona servicios compartidos que utilizan múltiples usuarios en la red. Como un servidor Internet, Windows puede permitir miles de conexiones web simultáneas.

ARQUITECTURA

La Figura 2.13 muestra la estructura global de Windows 2000; posteriores versiones de Windows tienen esencialmente la misma estructura a este nivel de detalle. Su estructura modular da a Windows una considerable flexibilidad. Se ha diseñado para ejecutar en una variedad de plataformas hardware y da soporte a aplicaciones escritas para una gran cantidad de sistemas operativos. En el momento de escritura de este libro, Windows está implementado solamente en las plataformas hardware Intel Pentium/x86 e Itanium.

Como virtualmente en todos los sistemas operativos, Windows separa el software orientado a aplicación del software del sistema operativo. La segunda parte, que incluye el sistema ejecutivo, el

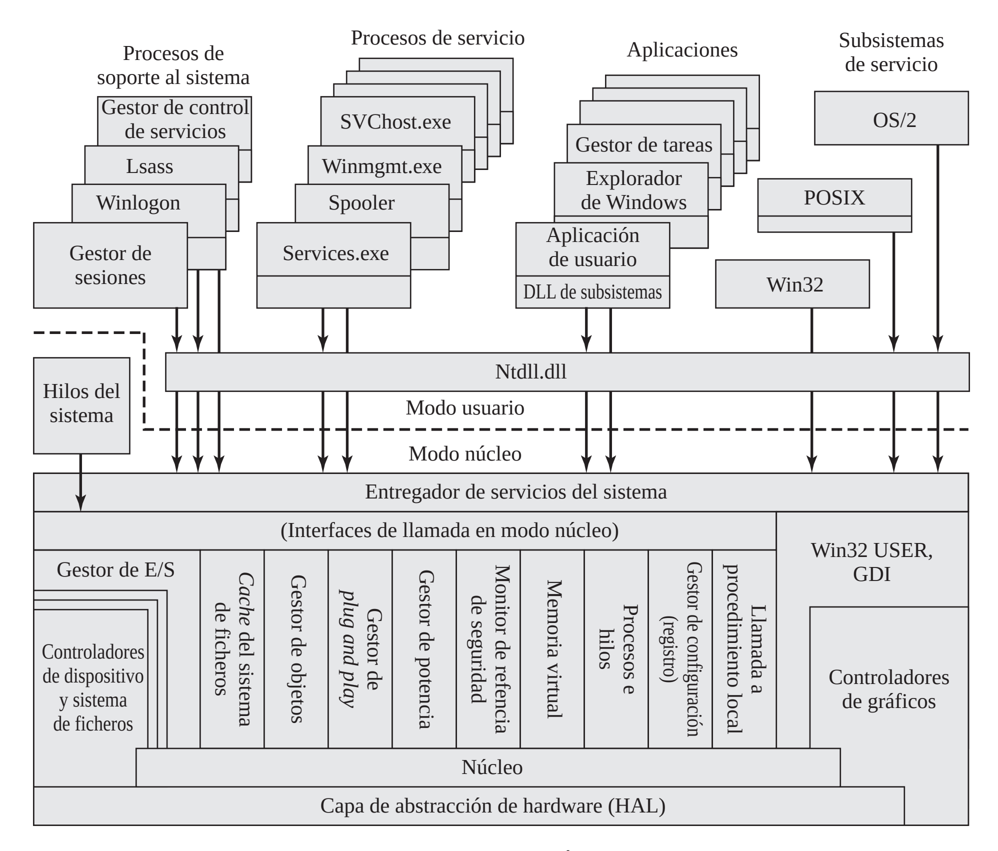


Figura 2.13. Arquitectura de Windows 2000 [SOLO00].

núcleo o *kernel* y la capa de abstracción del hardware, ejecuta en modo núcleo. El software que ejecuta en modo núcleo tiene acceso a los datos del sistema y al hardware. El resto del software, que ejecuta en modo usuario, tiene un acceso limitado a los datos del sistema.

ORGANIZACIÓN DEL SISTEMA OPERATIVO

Windows no tiene una arquitectura micronúcleo pura, sino lo que Microsoft denomina arquitectura micronúcleo modificada. Como en el caso de las arquitecturas micronúcleo puras, Windows es muy modular. Cada función del sistema se gestiona mediante un único componente del sistema operativo. El resto del sistema operativo y todas las aplicaciones acceden a dicha función a través del componente responsable y utilizando una interfaz estándar. Sólo se puede acceder a los datos del sistema claves mediante la función apropiada. En principio, se puede borrar, actualizar o reemplazar cualquier módulo sin reescribir el sistema completo o su interfaz de programa de aplicación (API). Sin embargo, a diferencia de un sistema micronúcleo puro, Windows se configura de forma que muchas de las funciones del sistema externas al micronúcleo ejecutan en modo núcleo. La razón reside en el rendimiento. Los desarrolladores de Windows descubrieron que utilizando la técnica micronúcleo pura, muchas funciones fuera del micronúcleo requerían varios intercambios entre procesos o hilos, cambios de modo y el uso de *buffers* de memoria extra. Los componentes en modo núcleo son los siguientes:

- **Sistema ejecutivo.** Contiene los servicios básicos del sistema operativo, como la gestión de memoria, la gestión de procesos e hilos, seguridad, E/S y comunicación entre procesos.
- **Núcleo.** Está formado por los componentes más fundamentales del sistema operativo. El núcleo gestiona la planificación de hilos, el intercambio de procesos, las excepciones, el manejo de interrupciones y la sincronización de multiprocesadores. A diferencia del resto del sistema ejecutivo y el nivel de usuario, el código del núcleo no se ejecuta en hilos. Por tanto, es la única parte del sistema operativo que no es expulsable o paginable.
- **Capa de abstracción de hardware (HAL: *Hardware Abstraction Layer*).** Realiza una proyección entre mandatos y respuestas hardware genéricos y aquéllos que son propios de una plataforma específica. Aísla el sistema operativo de las diferencias de hardware específicas de la plataforma. El HAL hace que el bus de sistema, el controlador de acceso a memoria directa (DMA), el controlador de interrupciones, los temporizadores de sistema y los módulos de memoria de cada máquina parezcan los mismos al núcleo. También entrega el soporte necesario para multiprocesamiento simétrico (SMP), explicado anteriormente.
- **Controladores de dispositivo.** Incluye tanto sistemas de ficheros como controladores de dispositivos hardware que traducen funciones de E/S de usuario en peticiones específicas a dispositivos hardware de E/S.
- **Gestión de ventanas y sistemas gráficos.** Implementa las funciones de la interfaz gráfica de usuario (GUI), tales como la gestión de ventanas, los controles de la interfaz de usuario y el dibujo.

El sistema ejecutivo de Windows incluye módulos para funciones del sistema específicas y proporciona un API para software en modo usuario. A continuación se describen cada uno de estos módulos del sistema ejecutivo:

- **Gestor de E/S.** Proporciona un entorno a través del cual las aplicaciones pueden acceder a los dispositivos de E/S. El gestor de E/S es responsable de enviar la petición al controlador del dispositivo apropiado para un procesamiento posterior. El gestor de E/S implementa todas las API de E/S de Windows y provee seguridad y nombrado para dispositivos y sistemas de ficheros (utilizando el gestor de objetos). La E/S de Windows se discute en el Capítulo 11.

- **Gestor de cache.** Mejora el rendimiento de la E/S basada en ficheros, provocando que los datos de disco referenciados recientemente residan en memoria principal para un acceso rápido, y retardando las escrituras en disco a través del mantenimiento de las actualizaciones en memoria durante un periodo corto de tiempo antes de enviarlas a disco.
- **Gestor de objetos.** Crea, gestiona y borra los objetos del sistema ejecutivo de Windows y los tipos de datos abstractos utilizados para representar recursos como procesos, hilos y objetos de sincronización. Provee reglas uniformes para mantener el control, el nombrado y la configuración de seguridad de los objetos. El gestor de objetos también crea los manejadores de objetos, que están formados por información de control de acceso y un puntero al objeto. Los objetos de Windows se discuten posteriormente en esta sección.
- **Gestor de *plug and play*.** Determina qué controladores se necesitan para un determinado dispositivo y carga dichos controladores.
- **Gestor de potencia.** Coordina la gestión de potencia entre varios dispositivos y se puede configurar para reducir el consumo de potencia hibernando el procesador.
- **Monitor de referencia de seguridad.** Asegura la validación de acceso y las reglas de generación de auditoría. El modelo orientado a objetos de Windows proporciona una visión de seguridad consistente y uniforme, especificando las entidades fundamentales que constituyen el sistema ejecutivo. Por tanto, Windows utiliza las mismas rutinas de validación de acceso y las comprobaciones de auditoria para todos los objetos protegidos, incluyendo ficheros, procesos, espacios de direcciones y dispositivos de E/S. La seguridad de Windows se discute en el Capítulo 15.
- **Gestor de memoria virtual.** Proyecta direcciones virtuales del espacio de direcciones del proceso a las páginas físicas de la memoria del computador. La gestión de memoria virtual de Windows se describe en el Capítulo 8.
- **Gestor de procesos e hilos.** Crea y borra los objetos y traza el comportamiento de los objetos proceso e hilo. La gestión de los procesos e hilos Windows se describe en el Capítulo 4.
- **Gestor de configuración.** Es responsable de implementar y gestionar el registro del sistema, que es el repositorio para la configuración de varios parámetros a nivel de sistema global y por usuario.
- **Utilidad de llamada a procedimiento local (LPC: *Local Procedure Call*).** Fuerza una relación cliente/servidor entre las aplicaciones y los subsistemas ejecutivos dentro de un único sistema, en un modo similar a una utilidad de llamada a procedimiento remoto (RPC: *Remote Procedure Call*) utilizada para procesamiento distribuido.

PROCESOS EN MODO USUARIO

Hay cuatro tipos básicos de procesos en modo usuario en Windows:

- **Procesos de sistema especiales.** Incluye servicios no proporcionados como parte del sistema operativo Windows, como el proceso de inicio y el gestor de sesiones.
- **Procesos de servicio.** Otros servicios de Windows como por ejemplo, el registro de eventos.
- **Subsistemas de entorno.** Expone los servicios nativos de Windows a las aplicaciones de usuario y por tanto, proporciona un entorno o personalidad de sistema operativo. Los subsistemas soportados son Win32, Posix y OS/2. Cada subsistema de entorno incluye bibliotecas de enlace dinámico (*Dynamic Link Libraries*, DLL), que convierten las llamadas de la aplicación de usuario a llamadas Windows.
- **Aplicaciones de usuario.** Pueden ser de cinco tipos: Win32, Posix, OS/2, Windows 3.1 o MS-DOS.

Windows está estructurado para soportar aplicaciones escritas para Windows 2000 y versiones posteriores. Windows 98 y varios sistemas operativos Windows proporcionan este soporte utilizando un solo sistema ejecutivo compacto a través de subsistemas de entorno protegidos. Los subsistemas protegidos son aquellas partes de Windows que interactúan con el usuario final. Cada subsistema es un proceso separado, y el sistema ejecutivo protege su espacio de direcciones del resto de subsistemas y aplicaciones. Un subsistema protegido proporciona una interfaz de usuario gráfica o de línea de mandatos que define el aspecto del sistema operativo para un usuario. Adicionalmente, cada subsistema protegido proporciona el API para dicho entorno operativo particular.

Esto significa que las aplicaciones creadas para un entorno operativo particular podrían ejecutarse sin ningún cambio en Windows, porque la interfaz del sistema operativo que ven es la misma que aquélla para la que se han escrito. De esta forma, por ejemplo, las aplicaciones basadas en OS/2 se pueden ejecutar en el sistema operativo Windows sin ninguna modificación. Más aún, debido a que el sistema Windows está diseñado para ser independiente de plataforma, mediante el uso de la capa de abstracción de hardware (HAL), debería ser relativamente fácil portar tanto los subsistemas protegidos como las aplicaciones soportadas de una plataforma hardware a otra. En muchos casos, sólo se requiere recompilar.

El subsistema más importante es Win32. Win32 es el API implementada tanto en Windows 2000 y versiones posteriores como en Windows 98. Algunas de las características de Win32 no están disponibles en Windows 98, pero las características implementadas en Windows 98 son idénticas a aquéllas de Windows 2000 y posteriores versiones.

MODELO CLIENTE/SERVIDOR

El sistema ejecutivo, los subsistemas protegidos y las aplicaciones se estructuran de acuerdo al modelo de computación cliente/servidor, que es un modelo común para la computación distribuida, que se discute en la sexta parte. Esta misma arquitectura se puede adoptar para el uso interno de un solo sistema, como es el caso de Windows.

Cada subsistema de entorno y subsistema del servicio ejecutivo se implementa como uno o más procesos. Cada proceso espera la solicitud de un cliente por uno de sus servicios (por ejemplo, servicios de memoria, servicios de creación de procesos o servicios de planificación de procesadores). Un cliente, que puede ser un programa u otro módulo del sistema operativo, solicita un servicio a través del envío de un mensaje. El mensaje se encamina a través del sistema ejecutivo al servidor apropiado. El servidor lleva a cabo la operación requerida y devuelve los resultados o la información de estado por medio de otro mensaje, que se encamina de vuelta al cliente mediante el servicio ejecutivo.

Las ventajas de la arquitectura cliente/servidor son las siguientes:

- Simplifica el sistema ejecutivo. Es posible construir diversos API sin conflictos o duplicaciones en el sistema ejecutivo. Se pueden añadir fácilmente nuevas interfaces.
- Mejora la fiabilidad. Cada módulo de los servicios ejecutivos se ejecuta como un proceso separado, con su propia partición de memoria, protegida de otros módulos. Además, los clientes no pueden acceder directamente al hardware o modificar la zona de memoria en la cual se almacena el sistema ejecutivo. Un único servidor puede fallar sin provocar el fallo o corromper el resto del sistema operativo.
- Proporciona a las aplicaciones maneras uniformes de comunicarse con el sistema ejecutivo a través de los LPC sin restringir la flexibilidad. Las aplicaciones cliente esconden el proceso de paso de mensajes a través de resguardos de funciones, que son contenedores no ejecutables almacenados en bibliotecas de enlace dinámicas (*Dynamic Link Libraries*, DLL). Cuando una

aplicación realiza una llamada a la interfaz del subsistema de entorno, el resguardo de la aplicación cliente empaqueta los parámetros de la llamada y los envía como un mensaje a un subsistema servidor que implementa la llamada.

- Proporciona una base adecuada para la computación distribuida. Normalmente, la computación distribuida utiliza el modelo cliente/servidor, con llamadas a procedimientos remotos implementadas utilizando módulos distribuidos cliente y servidor y el intercambio de mensajes entre clientes y servidores. Con Windows, un servidor local puede pasar un mensaje al servidor remoto para realizar su procesamiento en nombre de las aplicaciones locales cliente. Los clientes no necesitan saber si una petición es atendida local o remotamente. De hecho, si una petición se atiende de forma local o remota, puede cambiar dinámicamente, de acuerdo a las condiciones de carga actuales y a los cambios dinámicos de configuración.

HILOS Y SMP

Dos características importantes de Windows son el soporte que da a los hilos y a SMP, ambas características presentadas en la Sección 2.4. [CUST93] enumera las siguientes características de Windows que dan soporte a los hilos y a los SMP:

- Las rutinas del sistema operativo se pueden ejecutar en cualquier procesador disponible, y diferentes rutinas se pueden ejecutar simultáneamente en diferentes procesadores.
- Windows permite el uso de múltiple hilos de ejecución dentro de un único proceso. Múltiples hilos dentro del mismo proceso se pueden ejecutar en diferentes procesadores simultáneamente.
- Los procesos de servidor pueden utilizar múltiples hilos para procesar peticiones de más de un cliente simultáneamente.
- Windows proporciona mecanismos para compartir datos y recursos entre procesos y capacidades flexibles de comunicación entre procesos.

OBJETOS DE WINDOWS

Windows se apoya enormemente en los conceptos del diseño orientado a objetos. Este enfoque facilita la compartición de recursos y datos entre los procesos y la protección de recursos frente al acceso no autorizado. Entre los conceptos clave del diseño orientado a objetos utilizados por Windows se encuentran los siguientes:

- **Encapsulación.** Un objeto está compuesto por uno o más elementos de información, denominados atributos y uno o más procedimientos que se podrían llevar a cabo sobre esos datos, denominados servicios. La única forma de acceder a los datos en un objeto es invocando uno de los servicios del objeto. Por tanto, los datos de un objeto se pueden proteger fácilmente del uso no autorizado o incorrecto (por ejemplo, intentando ejecutar una pieza de datos no ejecutable).
- **Clases e instancias de objetos.** Una clase de objeto es una plantilla que lista los atributos y los servicios de un objeto y define ciertas características de los objetos. El sistema operativo puede crear instancias específicas de una clase de objetos cuando lo necesite. Por ejemplo, hay una única clase de objeto de proceso y un objeto de proceso por cada proceso actualmente activo. Este enfoque simplifica la creación y gestión de los objetos.
- **Herencia.** Esta característica no es soportada a nivel de usuario sino por alguna extensión dentro del sistema ejecutivo. Por ejemplo, los objetos directorio son ejemplos de objeto contenedor. Una propiedad de un objeto contenedor es que los objetos que contiene pueden heredar

propiedades del contenedor mismo. Como un ejemplo, supongamos que hay un directorio en el sistema de ficheros que está comprimido. Entonces, cualquier fichero que se cree dentro del contenedor directorio también estará comprimido.

- **Polimorfismo.** Internamente, Windows utiliza un conjunto común de funciones para manipular objetos de cualquier tipo; ésta es una característica de polimorfismo, tal y como se define en el Apéndice B. Sin embargo, Windows no es completamente polimórfico, porque hay muchas API que son específicas para tipos de objetos específicos.

El lector que no esté familiarizado con los conceptos orientados a objetos debe revisar el Apéndice B, que se encuentra al final del libro.

No todas las entidades de Windows son objetos. Los objetos se utilizan en casos donde los datos se usan en modo usuario y cuando el acceso a los datos es compartido o restringido. Entre las entidades representadas por los objetos se encuentran los ficheros, procesos, hilos, semáforos, temporizadores y ventanas. Windows crea y gestiona todos los tipos de objetos de una forma uniforme, a través del gestor de objetos. El gestor de objetos es responsable de crear y destruir objetos en nombre de las aplicaciones y de garantizar acceso a los servicios y datos de los objetos.

Cada objeto dentro del sistema ejecutivo, algunas veces denominado objeto del núcleo (para distinguirlo de los objetos a nivel de usuario, objetos que no son gestionados por el sistema ejecutivo), existe como un bloque de memoria gestionado por el núcleo y que es accesible solamente por el núcleo. Algunos elementos de la estructura de datos (por ejemplo, el nombre del objeto, parámetros de seguridad, contabilidad de uso) son comunes a todos los tipos de objetos, mientras que otros elementos son específicos de un tipo de objeto particular (por ejemplo, la prioridad del objeto hilo). Sólo el núcleo puede acceder a estas estructuras de datos de los objetos del núcleo; es imposible que una aplicación encuentre estas estructuras de datos y las lea o escriba directamente. En su lugar, las aplicaciones manipulan los objetos indirectamente a través del conjunto de funciones de manipulación de objetos soportado por el sistema ejecutivo. Cuando se crea un objeto, la aplicación que solicita la creación recibe un manejador del objeto. Esencialmente un manejador es un puntero al objeto referenciado. Cualquier hilo puede utilizar este manejador dentro del mismo proceso para invocar las funciones Win32 que trabajan con objetos.

Los objetos pueden tener información de seguridad asociada con ellos, en la forma de un descriptor de seguridad (*Security Descriptor*, SD). Esta información de seguridad se puede utilizar para restringir el acceso al objeto. Por ejemplo, un proceso puede crear un objeto semáforo con el objetivo de que sólo ciertos usuarios deben ser capaces de abrir y utilizar el semáforo. El SD de dicho objeto semáforo puede estar compuesto por la lista de aquellos usuarios que pueden (o no pueden) acceder al objeto semáforo junto con el conjunto de accesos permitidos (lectura, escritura, cambio, etc.).

En Windows, los objetos pueden tener nombre o no. Cuando un proceso crea un objeto sin nombre, el gestor de objetos devuelve un manejador para dicho objeto, y el manejador es la única forma de referirse a él. Los objetos con nombre son referenciados por otros procesos mediante dicho nombre. Por ejemplo, si un proceso A desea sincronizarse con el proceso B podría crear un objeto de tipo evento con nombre y pasar el nombre del evento a B. El proceso B podría entonces abrir y utilizar el objeto evento. Sin embargo, si A simplemente deseara utilizar el evento para sincronizar dos hilos dentro del proceso, crearía un objeto evento sin nombre, porque no necesita que otros procesos puedan utilizar dicho evento.

Como ejemplo de los objetos gestionados por Windows, a continuación se listan las dos categorías de objetos que gestiona el núcleo:

- **Objetos de control.** Utilizados para controlar las operaciones del núcleo en áreas que no corresponden a la planificación y la sincronización. La Tabla 2.5 lista los objetos de control del núcleo.

- **Objetos *dispatcher*.** Controla la activación y la sincronización de las operaciones del sistema. Estos objetos se describen en el Capítulo 6.

Tabla 2.5. Objetos de control del micronúcleo de Windows [MS96].

Llamada a procedimiento asíncrono	Utilizado para romper la ejecución de un hilo específico y provocar que se llame a un procedimiento en un modo de procesador especificado
Interrupción	Utilizado para conectar un origen de interrupción a una rutina de servicio de interrupciones por medio de una entrada en una tabla IDT (*Interrupt Dispatch Table*). Cada procesador tiene una tabla IDT que se utiliza para entregar interrupciones que ocurren en dicho procesador.
Proceso	Representa el espacio de direcciones virtuales e información de control necesaria para la ejecución de un conjunto de objetos hilo. Un proceso contiene un puntero a un mapa de direcciones, una lista de hilos listos para ejecutar que contiene objetos hilo, una lista de hilos que pertenecen al proceso, el tiempo total acumulado para todos los hilos que se ejecuten dentro del proceso y una prioridad base.
Perfil	Utilizado para medir la distribución de tiempo de ejecución dentro de un bloque de código. Se puede medir el perfil tanto de código de usuario como de sistema.

Windows no es un sistema operativo completamente orientado a objetos. No está implementado en un lenguaje orientado a objetos. Las estructuras de datos que residen completamente dentro de un componente del sistema ejecutivo no están representadas como objetos. Sin embargo, Windows muestra la potencia de la tecnología orientada a objetos y representa la tendencia cada vez más significativa del uso de esta tecnología en el diseño de los sistemas operativos.

2.6. SISTEMAS UNIX TRADICIONALES

HISTORIA

La historia de UNIX es un relato narrado con frecuencia y no se repetirá con muchos detalles aquí. Por el contrario, aquí se realizará un breve resumen.

UNIX se desarrolló inicialmente en los laboratorios Bell y se hizo operacional en un PDP-7 en 1970. Algunas de las personas relacionadas con los laboratorios Bell también habían participado en el trabajo de tiempo compartido desarrollado en el proyecto MAC del MIT. Este proyecto se encargó primero del desarrollo de CTSS y después de Multics. Aunque es habitual decir que el UNIX original fue una versión recortada de Multics, los desarrolladores de UNIX realmente dijeron estar más influenciados por CTSS [RITC78]. No obstante, UNIX incorporó muchas ideas de Multics.

El trabajo de UNIX en los laboratorios Bell, y después en otras instituciones, produjo un conjunto de versiones de UNIX. El primer hito más notable fue portar el sistema UNIX de PDP-7 al PDP-11. Ésta fue la primera pista de que UNIX sería un sistema operativo para todos los computadores. El siguiente hito importante fue la reescritura de UNIX en el lenguaje de programación C. Ésta era una estrategia sin precedentes en ese tiempo. Se pensaba que algo tan complejo como un sistema operativo, que debe tratar eventos críticos en el tiempo, debía escribirse exclusivamente en lenguaje ensamblador. La implementación C demostró las ventajas de utilizar un lenguaje de alto nivel para la mayoría o todo el código del sistema. Hoy, prácticamente todas las implementaciones UNIX están escritas en C.

Estas primeras versiones de UNIX fueron populares dentro de los laboratorios Bell. En 1974, el sistema UNIX se describió en una revista técnica por primera vez [RITC74]. Esto despertó un gran interés por el sistema. Se proporcionaron licencias de UNIX tanto a instituciones comerciales como a universidades. La primera versión completamente disponible fuera de los laboratorios Bell fue la Versión 6, en 1976. La siguiente versión, la Versión 7, aparecida en 1978, es la antecesora de los sistemas UNIX más modernos. El sistema más importante no vinculado con AT&T se desarrolló en la Universidad de California en Berkeley, y se llamó UNIX BSD (*Berkeley Software Distribution*), ejecutándose primero en PDP y después en máquinas VAX. AT&T continuó desarrollando y refinando el sistema. En 1982, los laboratorios Bell combinaron diversas variantes AT&T de UNIX en un único sistema, denominado comercialmente como UNIX System III. Un gran número de características se añadieron posteriormente al sistema operativo para producir UNIX System V.

DESCRIPCIÓN

La Figura 2.14 proporciona una descripción general de la arquitectura UNIX. El hardware subyacente es gestionado por el software del sistema operativo. El sistema operativo se denomina frecuentemente el núcleo del sistema, o simplemente núcleo, para destacar su aislamiento frente a los usuarios y a las aplicaciones. Esta porción de UNIX es lo que se conocerá como UNIX en este libro. Sin embargo, UNIX viene equipado con un conjunto de servicios de usuario e interfaces que se consideran parte del sistema. Estos se pueden agrupar en el *shell*, otro software de interfaz, y los componentes del compilador C (compilador, ensamblador, cargador). La capa externa está formada por las aplicaciones de usuario y la interfaz de usuario al compilador C.

La Figura 2.15 muestra una vista más cercana del núcleo. Los programas de usuario pueden invocar los servicios del sistema operativo directamente o a través de programas de biblioteca. La interfaz de llamada a sistemas es la frontera con el usuario y permite que el software de alto nivel obtenga acceso a funciones específicas de núcleo. En el otro extremo, el sistema operativo contiene rutinas primitivas que interaccionan directamente con el hardware. Entre estas dos interfaces, el sistema se divi-

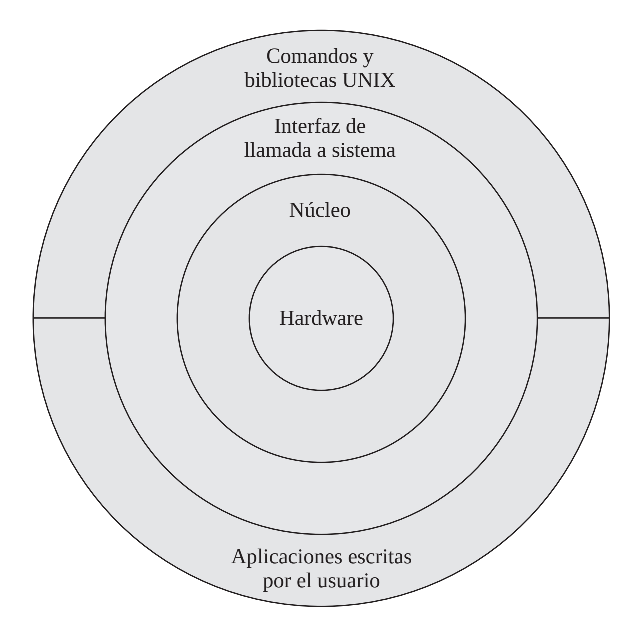


Figura 2.14. Arquitectura general de UNIX.

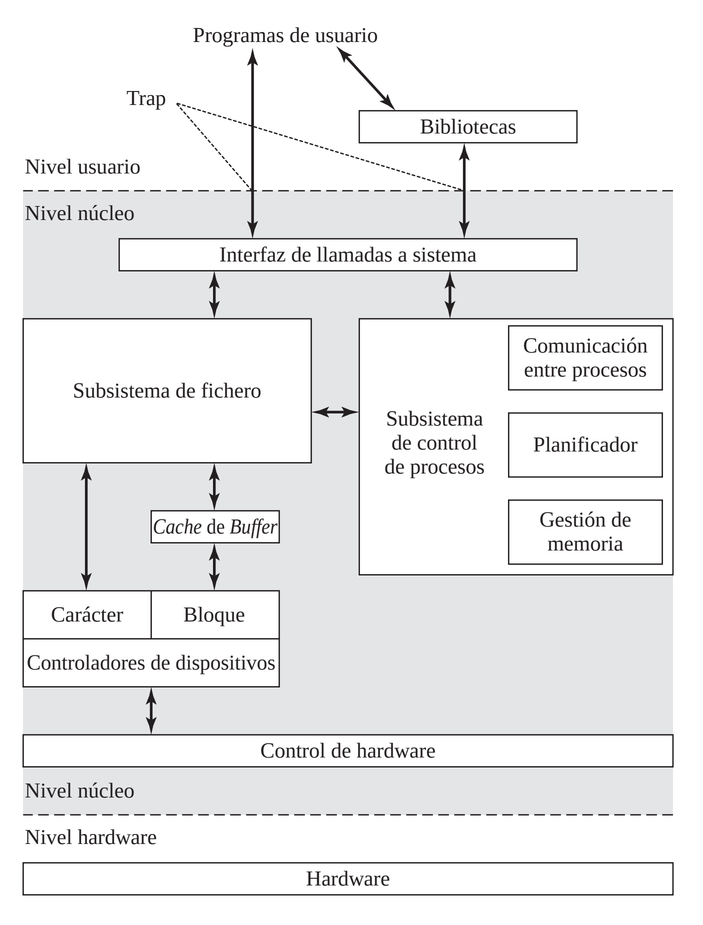


Figura 2.15. Núcleo tradicional de UNIX [BACH86].

de en dos partes principales, una encargada del control de procesos y la otra encargada de la gestión de ficheros y de la E/S. El subsistema de control de procesos se encarga de la gestión de memoria, la planificación y ejecución de los procesos, así como de la sincronización y la comunicación entre los procesos. El sistema de ficheros intercambia datos entre la memoria y los dispositivos externos tanto como flujos de caracteres como bloques. Para lograr esto, se utilizan una gran variedad de controladores de dispositivos. Para las transferencias orientadas a bloques, se utiliza una técnica de cache de discos: entre el espacio de direccionamiento del usuario y el dispositivo externo se interpone un *buffer* de sistema en memoria principal.

La descripción de esta subsección se refiere a lo que se han denominado sistemas UNIX tradicionales; [VAHA96] utiliza este término para referirse a System V Versión 3 (SVR3: *System V Release 3*), 4.3BSD y versiones anteriores. Las siguientes sentencias generales pueden afirmarse sobre un sistema UNIX tradicional. Se diseñaron para ejecutar sobre un único procesador y carecen de la capacidad para proteger sus estructuras de datos del acceso concurrente por parte de múltiples procesadores. Su núcleo no es muy versátil, soportando un único tipo de sistema de ficheros, una única política de planificación de procesos y un único formato de fichero ejecutable. El núcleo tradicional de UNIX no está diseñado para ser extensible y tiene pocas utilidades para la reutilización de código. El resultado es que, según se iban añadiendo nuevas características a varias versiones de UNIX, se tuvo que añadir mucho código, proporcionando un núcleo de gran tamaño y no modular.

2.7. SISTEMAS UNIX MODERNOS

Cuando UNIX evolucionó, un gran número de diferentes implementaciones proliferó, cada una de las cuales proporcionó algunas características útiles. Fue necesaria la producción de una nueva implementación que unificara muchas de las importantes innovaciones, añadiera otras características de diseño de los sistemas operativos modernos, y produjera una arquitectura más modular. La arquitectura mostrada en la Figura 2.16 muestra los aspectos típicos de un núcleo UNIX moderno. Existe un pequeño núcleo de utilidades, escritas de forma modular, que proporciona funciones y servicios necesarios para procesos del sistema operativo. Cada uno de los círculos externos representa funciones y una interfaz que podría implementarse de diferentes formas.

Ahora se verán algunos ejemplos de sistemas UNIX modernos.

SYSTEM V RELEASE 4 (SVR4)

SVR4, desarrollado conjuntamente por AT&T y Sun Microsistemas, combina características de SVR3, 4.3BSD, Microsoft Xenix System y SunOS. Fue casi una reescritura completa del núcleo del

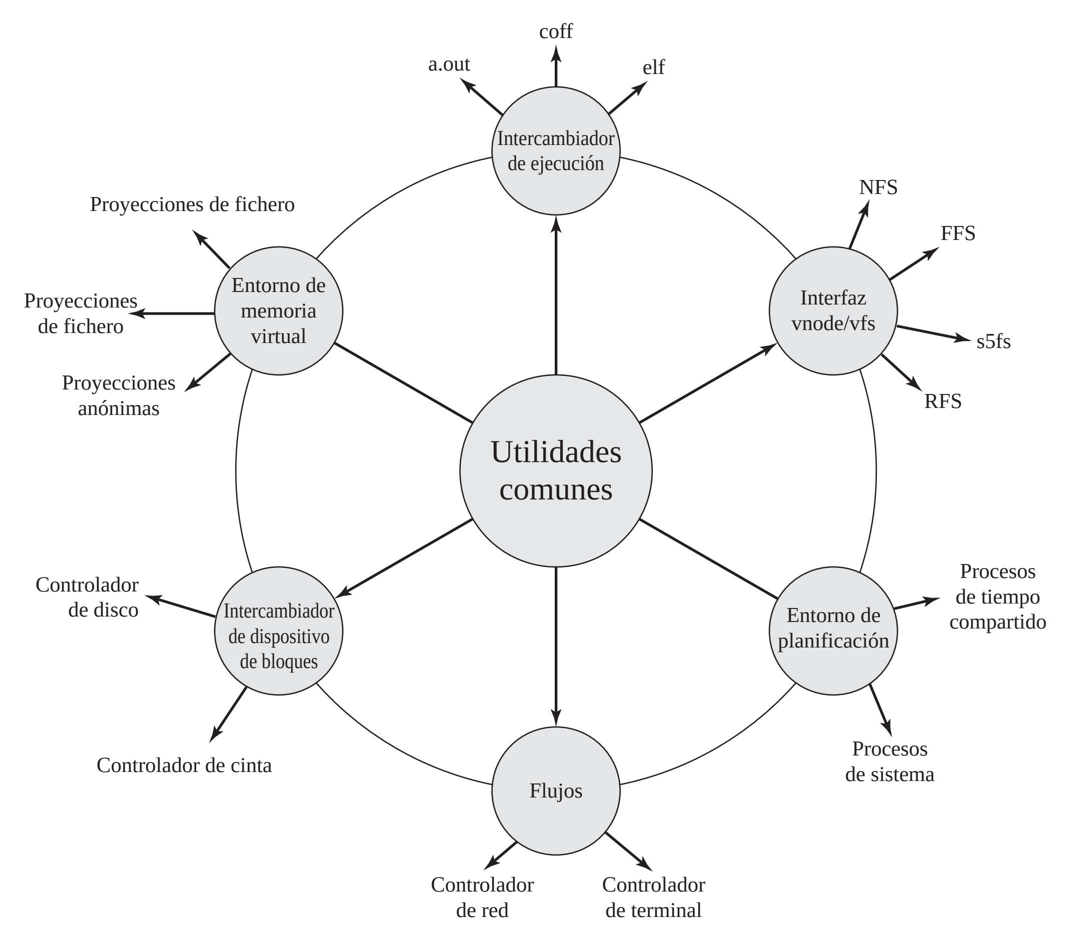


Figura 2.16. Núcleo UNIX moderno [VAHA96].

System V y produjo una implementación bien organizada, aunque compleja. Las nuevas características de esta versión incluyen soporte al procesamiento en tiempo real, clases de planificación de procesos, estructuras de datos dinámicamente asignadas, gestión de la memoria virtual, sistema de ficheros virtual y un núcleo expulsivo.

SVR4 mezcló los esfuerzos de los diseñadores comerciales y académicos y se desarrolló para proporcionar una plataforma uniforme que permitiera el despegue comercial de UNIX. Logró su objetivo y es quizá una de las variantes más importantes de UNIX. Incorpora la mayoría de las características importantes desarrolladas en cualquier sistema UNIX y lo hace de una forma integrada y comercialmente viable. SVR4 ejecuta en un gran rango de máquinas, desde los microprocesadores de 32 bits hasta los supercomputadores. Muchos de los ejemplos UNIX de este libro son ejemplos de SVR4.

SOLARIS 9

Solaris es una versión UNIX de Sun basada en SVR4. La última versión es la 9, y proporciona todas las características de SVR4 más un conjunto de características avanzadas, como un núcleo multihilo, completamente expulsivo, con soporte completo para SMP, y una interfaz orientada a objetos para los sistemas de ficheros. Solaris es la implementación UNIX más utilizada y comercialmente más exitosa. Para algunas características de los sistemas operativos, se utiliza a Solaris como ejemplo en este libro.

4.4BSD

Las series de UNIX BSD (*Berkeley Software Distribution*) han jugado un papel importante en el desarrollo de la teoría de diseño de los sistemas operativos. 4.xBSD se ha utilizado ampliamente en instalaciones académicas y ha servido como base de algunos productos comerciales UNIX. Es probable que BSD es seguramente responsable de gran parte de la popularidad de UNIX y que la mayoría de las mejoras de UNIX aparecieron en primer lugar en las versiones BSD.

4.4BSD fue la versión final de BSD que Berkeley produjo, disolviéndose posteriormente la organización encargada del diseño e implementación. Se trata de una actualización importante de 4.3BSD, que incluye un nuevo sistema de memoria virtual, cambios en la estructura del núcleo, y una larga lista de otras mejoras.

La última versión del sistema operativo de Macintosh, Mac OS X, se basa en 4.4BSD.

2.8. LINUX

HISTORIA

Linux comenzó como una variante UNIX para la arquitectura del PC IBM (Intel 80386). Linus Torvalds, un estudiante finlandés de informática, escribió la versión inicial. Torvalds distribuyó por Internet una primera versión de Linux en 1991. Desde entonces, algunas personas, colaborando en Internet, han contribuido al desarrollo de Linux, todo bajo el control de Torvalds. Debido a que Linux es libre y el código fuente está disponible, se convirtió pronto en una alternativa para otras estaciones de trabajo UNIX, tal como las ofrecidas por Sun Microsystems e IBM. Hoy en día, Linux es un sistema UNIX completo que ejecuta en todas esas plataformas y algunas más, incluyendo Intel Pentium e Itanium, y el PowerPC de Motorota/IBM.

La clave del éxito de Linux ha sido la disponibilidad de los paquetes de software libre bajo los auspicios de la Fundación de Software Libre (*Free Software Foundation*, FSF). Esta fundación se centra en un software estable, independiente de plataforma, con alta calidad, y soportado por la comunidad de usuarios. El proyecto de GNU proporciona herramientas para desarrolladores de software y la licencia pública GNU (GPL: *GNU Public License*) es el sello de aprobación de FSF. Torvald utilizó herramientas GNU para el desarrollo del núcleo, que fue posteriormente distribuido bajo la licencia GPL. Por tanto, las distribuciones Linux que aparecen hoy en día son los productos del proyecto GNU de FSF, los esfuerzos individuales de Torvalds y muchos colaboradores a lo largo del mundo.

Además de su uso por muchos programadores individuales, Linux ha hecho ahora una penetración significativa en el mundo corporativo. Esto no es sólo debido al software libre, sino también a la calidad del núcleo de Linux. Muchos programadores con talento han contribuido a la versión actual, dando lugar a un producto técnicamente impresionante. Más aún, Linux es muy modular y fácilmente configurable. Resulta óptimo para incrementar el rendimiento de una variedad de plataformas hardware. Además, con el código fuente disponible, los distribuidores pueden adaptar las aplicaciones y facilidades para cumplir unos requisitos específicos. A lo largo de este libro, se proporcionarán detalles internos del núcleo de Linux.

ESTRUCTURA MODULAR

La mayoría de los núcleos Linux son monolíticos. Como se mencionó anteriormente en el capítulo, un núcleo monolítico es aquél que incluye prácticamente toda la funcionalidad del sistema operativo en un gran bloque de código que ejecuta como un único proceso con un único espacio de direccionamiento. Todos los componentes funcionales del núcleo tienen acceso a todas las estructuras internas de datos y rutinas. Si los cambios se hacen sobre cualquier porción de un sistema operativo monolítico, todos los módulos y rutinas deben volverse a enlazar y reinstalar, y el sistema debe ser reiniciado para que los cambios tengan efecto. Como resultado, cualquier modificación, como por ejemplo añadir un nuevo controlador de dispositivo o función del sistema de fichero, es difícil. Este problema es especialmente agudo para Linux, cuyo desarrollo es global y ha sido realizado por un grupo de programadores independientes asociados de forma difusa.

Aunque Linux no utiliza una técnica de micronúcleo, logra muchas de las ventajas potenciales de esta técnica por medio de su arquitectura modular particular. Linux está estructurado como una colección de módulos, algunos de los cuales pueden cargarse y descargarse automáticamente bajo demanda. Estos bloques relativamente independientes se denominan **módulos cargables** [GOYE99]. Esencialmente, un módulo es un fichero cuyo código puede enlazarse y desenlazarse con el núcleo en tiempo real. Normalmente, un módulo implementa algunas funciones específicas, como un sistema de ficheros, un controlador de dispositivo o algunas características de la capa superior del núcleo. Un módulo no se ejecuta como su propio proceso o hilo, aunque puede crear los hilos del núcleo que necesite por varios propósitos. En su lugar, un módulo se ejecuta en modo núcleo en nombre del proceso actual.

Por tanto, aunque Linux se puede considerar monolítico, su estructura modular elimina algunas de las dificultades para desarrollar y evolucionar el núcleo.

Los módulos cargables de Linux tienen dos características importantes:

- **Enlace dinámico.** Un módulo de núcleo puede cargarse y enlazarse al núcleo mientras el núcleo está en memoria y ejecutándose. Un módulo también puede desenlazarse y eliminarse de la memoria en cualquier momento.

- **Módulos apilables.** Los módulos se gestionan como una jerarquía. Los módulos individuales sirven como bibliotecas cuando los módulos cliente los referencian desde la parte superior de la jerarquía, y actúan como clientes cuando referencian a módulos de la parte inferior de la jerarquía.

El enlace dinámico [FRAN97] facilita la configuración y reduce el uso de la memoria del núcleo. En Linux, un programa de usuario o un usuario puede cargar y descargar explícitamente módulos del núcleo utilizando los mandatos *insmod* y *rmmod*. El núcleo mismo detecta la necesidad de funciones particulares y puede cargar y descargar módulos cuando lo necesite. Con módulos apilables, se pueden definir dependencias entre los módulos. Esto tiene dos ventajas:

1. El código común para un conjunto de módulos similares (por ejemplo, controladores para hardware similar) se puede mover a un único módulo, reduciendo la replicación.
2. El núcleo puede asegurar que los módulos necesarios están presentes, impidiendo descargar un módulo del cual otros módulos que ejecutan dependen y cargando algunos módulos adicionalmente requeridos cuando se carga un nuevo módulo.

La Figura 2.17 es un ejemplo que ilustra las estructuras utilizadas por Linux para gestionar módulos. La figura muestra la lista de los módulos del núcleo que existen después de que sólo dos módulos han sido cargados: FAT y VFAT. Cada módulo se define mediante dos tablas, la tabla de módulos y la tabla de símbolos. La tabla de módulos incluye los siguientes elementos:

- *next. Puntero al siguiente módulo. Todos los módulos se organizan en una lista enlazada. La lista comienza con un pseudomódulo (no mostrado en la Figura 2.17).
- *name. Puntero al nombre del módulo.
- size. Tamaño del módulo en páginas de memoria
- usecount. Contador del uso del módulo. El contador se incrementa cuando una operación relacionada con las funciones del módulo comienza y se decrementa cuando la operación finaliza.
- flags. Opciones del módulo.
- nsyms. Número de símbolos exportados.
- ndeps. Número de módulos referenciados.
- *syms. Puntero a la tabla de símbolos de este módulo.
- *deps. Puntero a la lista de módulos referenciados por este módulo.
- *refs. Puntero a la lista de módulos que usa este módulo.

La tabla de símbolos define aquellos símbolos controlados por este módulo que se utilizan en otros sitios.

La Figura 2.17 muestra que el módulo VFAT se carga después del módulo FAT y que el módulo VFAT es dependiente del módulo FAT.

COMPONENTES DEL NÚCLEO

La Figura 2.18, tomada de [MOSB02] muestra los principales componentes del núcleo Linux tal y como están implementados en una arquitectura IA-64 (por ejemplo, Intel Itanium). La figura muestra

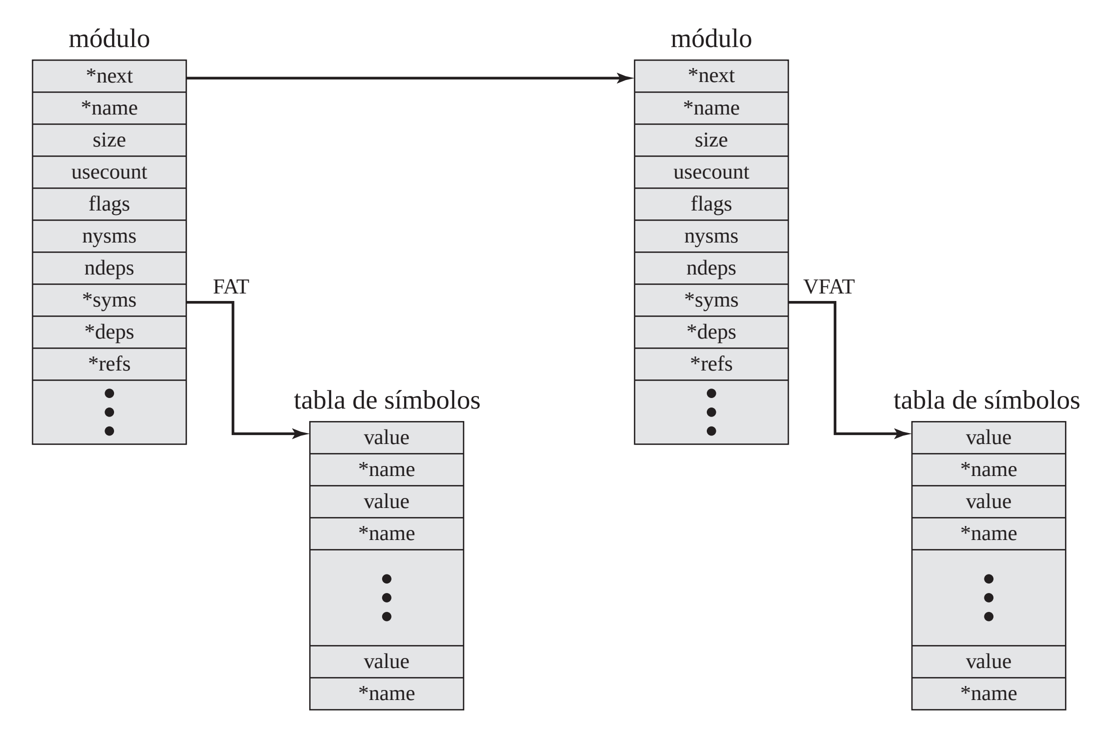


Figura 2.17. Lista ejemplo de módulos de núcleo de Linux.

varios procesos ejecutando encima del núcleo. Cada caja indica un proceso separado, mientras que cada línea curvada con una cabeza de flecha representa un hilo de ejecución*.

El núcleo mismo está compuesto por una colección de componentes que interaccionan, usando flechas para indicar las principales interacciones. También se muestra el hardware subyacente como un conjunto de componentes utilizando flechas para indicar qué componentes del núcleo utilizan o controlan qué componentes del hardware. Todos los componentes del núcleo, por supuesto, ejecutan en la CPU, pero por simplicidad no se muestran estas relaciones.

Brevemente, los principales componentes del núcleo son los siguientes:

- **Señales.** El núcleo utiliza las señales para llamar a un proceso. Por ejemplo, las señales se utilizan para notificar ciertos fallos a un proceso como por ejemplo, la división por cero. La Tabla 2.6 da unos pocos ejemplos de señales.
- **Llamadas al sistema.** La llamada al sistema es la forma en la cual un proceso requiere un servicio de núcleo específico. Hay varios cientos de llamadas al sistema, que pueden agruparse básicamente en seis categorías: sistema de ficheros, proceso, planificación, comunicación entre procesos, *socket* (red) y misceláneos. La Tabla 2.7 define unos pocos ejemplos de cada categoría.

En Linux, no hay distinción entre los conceptos de proceso e hilo. Sin embargo, múltiples hilos en Linux se pueden agrupar de tal forma que, efectivamente, pueda existir un único proceso compuesto por múltiples hilos. Estos aspectos se discuten en el Capítulo 4.

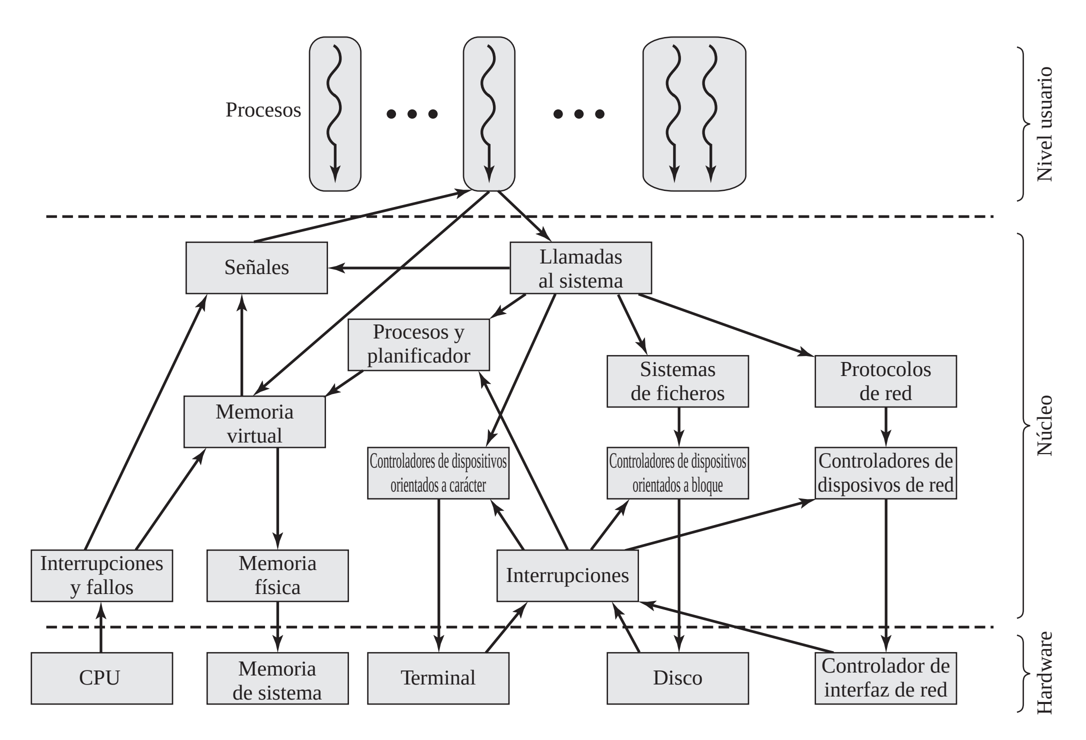


Figura 2.18. Componentes del núcleo de Linux.

- **Procesos y planificador.** Crea, gestiona y planifica procesos.
- **Memoria virtual.** Asigna y gestiona la memoria virtual para los procesos.
- **Sistemas de ficheros.** Proporciona un espacio de nombres global y jerárquico para los ficheros, directorios y otros objetos relacionados con los ficheros. Además, proporciona las funciones del sistema de ficheros.
- **Protocolos de red.** Da soporte a la interfaz *Socket* para los usuarios, utilizando la pila de protocolos TCP/IP.
- **Controladores de dispositivo tipo carácter.** Gestiona los dispositivos que requiere el núcleo para enviar o recibir datos un byte cada vez, como los terminales, los módems y las impresoras.
- **Controladores de dispositivo tipo bloque.** Gestiona dispositivos que leen y escriben datos en bloques, tal como varias formas de memoria secundaria (discos magnéticos, CDROM, etc.).
- **Controladores de dispositivo de red.** Gestiona las tarjetas de interfaz de red y los puertos de comunicación que permiten las conexiones a la red, tal como los puentes y los encaminadores.
- ***Traps* y fallos.** Gestiona los *traps* y fallos generados por la CPU, como los fallos de memoria.
- **Memoria física.** Gestiona el conjunto de marcos de páginas de memoria real y asigna las páginas de memoria virtual.
- **Interrupciones.** Gestiona las interrupciones de los dispositivos periféricos.

Tabla 2.6. Algunas señales de Linux.

SIGHUP	Desconexión de un terminal	SIGCONT	Continuar
SIGQUIT	Finalización por teclado	SIGTSTP	Parada por teclado
SIGTRAP	Traza	SIGTTOU	Escritura de terminal
SIGBUS	Error de bus	SIGXCPU	Límite de CPU excedido
SIGKILL	Señal para matar	SIGVTALRM	Reloj de alarma virtual
SIGSEGV	Violación de segmentación	SIGWINCH	Cambio de tamaño de una ventana
SIGPIPE	Tubería rota	SIGPWR	Fallo de potencia
SIGTERM	Terminación	SIGRTMIN	Primera señal de tiempo real
SIGCHLD	Cambio en el estado del hijo	SIGRTMAX	Última señal de tiempo real

Tabla 2.7. Algunas llamadas al sistema de Linux.

	Relacionadas con el sistema de ficheros
close	Cierra un descriptor de fichero.
link	Construye un nuevo nombre para un fichero.
open	Abre y posiblemente crea un fichero o dispositivo.
read	Lee un descriptor de fichero.
write	Escribe a través de un descriptor de fichero.
	Relacionadas con los procesos
execve	Ejecuta un programa.
exit	Termina el proceso que lo invoca.
getpid	Obtiene la identificación del proceso.
setuid	Establece la identidad del usuario del proceso actual.
prtrace	Proporciona una forma por la cual un proceso padre puede observar y controlar la ejecución de otro proceso, examinar y cambiar su imagen de memoria y los registros.
	Relacionadas con la planificación
sched_getparam	Establece los parámetros de planificación asociados con la política de planificación para el proceso identificado por su *pid*.
sched_get_priority_max	Devuelve el valor máximo de prioridad que se puede utilizar con el algoritmo de planificación identificado por la política.
sched_setscheduler	Establece tanto la política de planificación (por ejemplo, FIFO) como los parámetros asociados al *pid* del proceso.
sched_rr_get_interval	Escribe en la estructura *timespec* apuntada por el parámetro *tp* el cuanto de tiempo *round robin* para el proceso *pid*.
sched_yield	Un proceso puede abandonar el procesador voluntariamente sin necesidad de bloquearse a través de una llamada al sistema. El proceso entonces se moverá al final de la cola por su prioridad estática y un nuevo proceso se pondrá en ejecución.

	Relacionadas con la comunicación entre procesos (IPC)
msgrcv	Se asigna una estructura de *buffer* de mensajes para recibir un mensaje. Entonces, la llamada al sistema lee un mensaje de la cola de mensajes especificada por *msqid* en el *buffer* de mensajes nuevamente creado.
semctl	Lleva a cabo la operación de control especificada por *cmd* en el conjunto de semáforos *semid*.
semop	Lleva a cabo operaciones en determinados miembros del conjunto de semáforos *semid*.
shmat	Adjunta el segmento de memoria compartido identificado por *shmid* al segmento de datos del proceso que lo invoca.
shmctl	Permite al usuario recibir información sobre un segmento de memoria compartido, establecer el propietario, grupo y permisos de un segmento de memoria compartido o destruir un segmento.
	Relacionadas con los *sockets* (red)
bind	Asigna la dirección IP local y puerto para un *socket*. Devuelve 0 en caso de éxito y -1 en caso de error.
connect	Establece una conexión entre el *socket* dado y el *socket* asociado remoto con *sockaddr*.
gethostname	Devuelve el nombre de máquina local.
send	Envia los bytes que tiene el *buffer* apuntado por **msg* sobre el *socket* dado.
setsockopt	Envia las opciones sobre un *socket*.
	Misceláneos
create_module	Intenta crear una entrada del módulo cargable y reservar la memoria de núcleo que será necesario para contener el módulo.
fsync	Copia todas las partes en memoria de un fichero a un disco y espera hasta que el dispositivo informa que todas las partes están en almacenamiento estable.
query_module	Solicita información relacionada con los módulos cargables desde el núcleo.
time	Devuelve el tiempo en segundos desde 1 de enero de 1970.
vhangup	Simula la suspensión del terminal actual. Esta llamada sirve para que otros usuarios puedan tener un terminal «limpio» en tiempo de inicio.

2.9. LECTURAS Y SITIOS WEB RECOMENDADOS

Como en el área de arquitectura de computadores, existen muchos libros de sistemas operativos. [SILB04], [NUTT04] y [TANE01] cubren los principios básicos usando diversos sistemas operativos importantes como casos de estudio. [BRIN01] es una colección excelente de artículos que cubren los principales avances del diseño de los sistemas operativos a lo largo de los años.

Un tratamiento excelente de los aspectos internos de UNIX, que proporciona un análisis comparativo de un gran número de variantes, es [VAHA96]. Para UNIX SVR4, [GOOD94] proporciona un tratamiento definitivo, con amplios detalles técnicos. Para el sistema académicamente popular Berkeley UNIX 4.4BSD, [MCKU96] es altamente recomendado. [MAUR01] proporciona un buen trata-

miento de los aspectos internos de Solaris. Dos buenos tratamientos de los aspectos internos de Linux se recogen en [BOVE03] y [BAR00].

Aunque hay incontables libros sobre varias versiones de Windows, hay curiosamente poco material disponible sobre los aspectos internos de Windows. [SOLO00] proporciona un tratamiento excelente de los aspectos internos de Windows 2000 y gran parte de este material es válido para posteriores versiones. [BOSW03] cubre parte de los aspectos internos de Windows 2003.

BAR00 Bar, M. *Linux Internals*. New York, McGraw-Hill, 2000.

BOSW03 Boswell, W. *Inside Windows Server 2003*. Reading, MA:Addison-Wesley, 2003.

BOVE03 Bovet, D., y Cesati, M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 2003.

BRIN01 Brinch Hansen, P. *Classic Operating Systems: From Batch Processing to Distributed Systems.* New York: Springer-Verlag, 2001.

GOOD94 Goodheart, B., y Cox, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

MAUR01 Mauro, J., y McDougall, R. *Solaris Internals: Core Kernel Architecture*. Palo Alto, CA: Sun Microsystems Press, 2001.

MCKU96 McKusick, M.; Bostic, K.; Karels, M.; y Quartermain, J. *The Design and Implementation of the 4.4BSD UNIX Operating System*. Reading, MA: Addison-Wesley, 1996.

NUTT04 Nutt,G. *Operating System*. Reading, MA: Addison-Wesley, 2004.

SILB04 Silberschatz, A.; Galvin, P.; y Gagne,G. *Operating System Concepts with Java*. Reading, MA: Addison-Wesley, 2004.

SOLO00 Solomon, D. *Inside Microsoft Windows 2000*. Redmond, WA: Microsoft Press, 2000.

TANE01 Tanenbaum, A. *Modern Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 2001.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.



SITIOS WEB RECOMENDADOS

- **El centro de recursos del sistema operativo.** Una colección útil de documentos y artículos sobre un amplio rango de temas de sistemas operativos.
- **Revisión de los sistemas operativos.** Una extensa revisión de sistemas operativos comerciales, libres, de investigación y para aficionados.
- **Comparación técnica de los sistemas operativos.** Incluye una cantidad sustancial de información sobre una variedad de sistemas operativos.
- **Grupo especial de interés ACM sobre los sistemas operativos.** Información sobre publicaciones y conferencias SIGOPS.
- **Comité Técnico IEEE sobre los sistemas operativos y los entornos de aplicación.** Incluye un boletín en línea y enlaces a otros sitios.
- **La FAQ comp.os.research.** FAQ amplio y válido que cubre aspectos de diseño de los sistemas operativos.

- **Universo Guru UNIX.** Fuente excelente de información sobre UNIX.
- **Proyecto de documentación Linux.** El nombre describe el sitio.

2.10. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

Contexto de ejecución	Monitor	Procesamiento serie
Dirección física	Monitor residente	Proceso
Dirección real	Monoprogramación	*Round-robin* o turno rotatorio
Dirección virtual	Multihilo	Sistema *batch* o en lotes
Estado del proceso	Multiprocesamiento simétrico	Sistema *batch* o en lotes multiprogramado
Gestión de memoria	Multiprogramación	Sistema de tiempo compartido
Hilo	Multitarea	Sistema operativo
Interrupción	Núcleo	Trabajo
Instrucción privilegiada	Núcleo monolítico	Tarea
Lenguaje de control de trabajos	Planificación	Tiempo compartido
Micronúcleo	Procesamiento *batch* o en lotes	

CUESTIONES DE REPASO

2.1. ¿Cuáles son los tres objetivos de diseño de un sistema operativo?

2.2. ¿Qué es el núcleo de un sistema operativo?

2.3. ¿Qué es multiprogramación?

2.4. ¿Qué es un proceso?

2.5. ¿Cómo utiliza el sistema operativo el contexto de ejecución de un proceso?

2.6. Liste y explique brevemente cinco responsabilidades relacionadas con la gestión de almacenamiento de un sistema operativo típico.

2.7. Explique la distinción entre una dirección real y una dirección virtual.

2.8. Describa la técnica de planificación *round-robin* o turno rotatorio.

2.9. Explique la diferencia entre un núcleo monolítico y un micronúcleo.

2.10. ¿En qué consiste el uso de multihilos o *multithreading*?

PROBLEMAS

2.1. Supóngase que se tiene un computador multiprogramado en el cual cada trabajo tiene características idénticas. En un periodo de computación T, un trabajo gasta la mitad del tiempo en E/S y la otra mitad en actividad del procesador. Cada trabajo ejecuta un total de N

periodos. Asúmase que se utiliza una planificación *round-robin* simple, y que las operaciones de E/S se pueden solapar con las operaciones del procesador. Defina las siguientes cantidades:

- Tiempo de servicio = Tiempo real para completar un trabajo.
- Productividad = Número medio de trabajos completados en el periodo *T*.
- Utilización del procesador = Porcentaje de tiempo que el procesador está activo (no esperando).

Calcule estas cantidades para uno, dos y cuatro trabajos simultáneos, asumiendo que el periodo *T* se distribuye de las siguientes formas:

a) Primera mitad E/S, segunda mitad procesador.

b) Primero y cuarto cuartos E/S, segundo y tercer cuartos procesador.

2.2. Un programa limitado por la E/S es aquel que, si ejecuta solo, gasta más tiempo esperando operaciones de E/S que utilizando el procesador. Un programa limitado por el procesador es lo contrario. Supóngase un algoritmo de planificación a corto plazo que favorece a aquellos programas que han utilizado poco tiempo de procesador en el pasado reciente. Explique por qué este algoritmo favorece a los programas limitados por la E/S y no niega permanentemente el tiempo de procesador a los programas limitados por el procesador.

2.3. Contraste las políticas de planificación que se podrían utilizar cuando se intenta optimizar un sistema de tiempo compartido y aquéllas que se utilizan para optimizar un sistema en lotes multiprogramado.

2.4. ¿Cuál es el propósito de las llamadas al sistema y cómo se relacionan las llamadas al sistema con el sistema operativo y el concepto de modo dual (modo núcleo y modo usuario)?

2.5. En el sistema operativo de IBM, OS/390, uno de los módulos principales en el núcleo es el gestor de recursos del sistema (SRM: *System Resource Manager*). Este módulo es responsable de la asignación de recursos entre los espacios de direcciones (procesos). El SRM da a OS/390 un grado de sofisticación único entre los sistemas operativos. Ningún otro sistema operativo de *mainframe*, y ciertamente ningún otro tipo de sistema operativo, puede realizar las funciones llevadas a cabo por SRM. El concepto de recurso incluye al procesador, memoria real y canales de E/S. SRM acumula estadísticas pertenecientes a la utilización del procesador, canales y varias estructuras de datos clave. Su propósito es proporcionar un rendimiento óptimo basado en monitorización y análisis del rendimiento. La instalación proporciona varios objetivos de rendimiento y éstas sirven como guía al SRM, que modifica dinámicamente la instalación y las características de rendimiento de los trabajos basándose en la utilización del sistema. Adicionalmente, SRM proporciona informes que permiten al operador entrenado refinar las configuraciones y el establecimiento de parámetros para mejorar el servicio de usuario.

Este problema es un ejemplo de actividad SRM. La memoria real se divide en bloques de igual tamaño llamados marcos, de los cuales podría haber muchos miles. Cada marco puede contener un bloque de memoria virtual, conocido como página. SRM recibe el control aproximadamente 20 veces cada segundo, inspeccionando cada marco de página. Si la página no se ha referenciado o cambiado, un contador se incrementa por 1. A lo largo del tiempo, SRM hace la media de estos números para determinar el número de segundos medio que un marco de página en el sistema queda inalterable. ¿Cuál podría ser el propósito de esto y qué acción podría realizar SRM?

PARTE II

PROCESOS

La tarea fundamental de cualquier sistema operativo moderno es la gestión de procesos. El sistema operativo debe reservar recursos para los procesos, permitir a los mismos compartir e intercambiar información, proteger los recursos de cada uno de ellos del resto, y permitir la sincronización entre procesos. Para conseguir alcanzar estos requisitos, el sistema operativo debe mantener una estructura determinada para cada proceso que describa el estado y la propiedad de los recursos y que permite al sistema operativo establecer el control sobre los procesos.

En un monoprocesador multiprogramado, la ejecución de varios procesos se puede intercalar en el tiempo. En un multiprocesador, no sólo se intercala la ejecución de procesos, también es posible que haya múltiples procesos que se ejecuten de forma simultánea. Tanto la ejecución intercalada como de forma simultánea son tipos de concurrencia y llevan a que el sistema se enfrente a diferentes problemas, tanto en el ámbito de las aplicaciones de programador como en el de los sistemas operativos.

En muchos sistemas operativos actuales, la problemática de la gestión de procesos se encuentra ampliada por la introducción del concepto de hilo (*thread*). En un sistema multihilo, el concepto de proceso mantiene los atributos de la propiedad de recursos, mientras que los aspectos de la ejecución de múltiples flujos de instrucciones se encuentra relacionada con los hilos que ejecutan dentro de ese proceso.

ÍNDICE PARA LA PARTE DOS

CAPÍTULO 3. DESCRIPCIÓN Y CONTROL DE PROCESOS

El objetivo de los sistemas operativos tradicionales es la gestión de procesos. Cada proceso se encuentra, en un instante dado, en uno de los diferentes estados de ejecución, que incluyen Listo, Ejecutando, y Bloqueado. El sistema operativo sigue la traza de estos estados de ejecución y gestiona el movimiento de procesos entre los mismos. Con este fin el sistema operativo mantiene unas estructuras de datos complejas que describen cada proceso. El sistema operativo debe realizar las operaciones de planificación y proporcionar servicios para la compartición entre procesos y la sincronización. El Capítulo 3 repasa estas estructuras de datos y las técnicas utilizadas de forma habitual por los sistemas operativos para la gestión de procesos.

CAPÍTULO 4. HILOS, SMP, Y MICRONÚCLEOS

El Capítulo 4 cubre tres áreas características de los sistemas operativos contemporáneos que representan unos avances sobre el diseño de los sistemas operativos tradicionales. En muchos sistemas operativos, el concepto tradicional de proceso se ha dividido en dos partes: una de ellas que trata de la propiedad de los recursos (proceso) y otra que trata de la ejecución del flujo de instrucciones (hilo o *thread*). Un único proceso puede contener múltiples hilos. La organización multihilo proporciona ventajas en la estructuración de las aplicaciones y en su rendimiento. El Capítulo 4 también examina los multiprocesadores simétricos (SMP), que son sistemas que tienen múltiples procesadores, cada uno de los cuales es capaz de ejecutar cualquier aplicación o el código de sistema. La organización SMP mejora el rendimiento y la fiabilidad. SMP a menudo se usa en conjunto con la programación multihilo, pero aún sin ella proporciona unas importantes mejoras a nivel de rendimiento. Para finalizar el Capítulo 4 examina el concepto de micronúcleo, que es un estilo de diseño de sistemas operativos que minimiza la cantidad de código de sistema que se ejecuta en modo núcleo. Las ventajas de esta estrategia también se analizan.

CAPÍTULO 5. CONCURRENCIA. EXCLUSIÓN MUTUA Y SINCRONIZACIÓN

Dos temas centrales en los sistemas operativos modernos son las multiprogramación y el procesamiento distribuido. El concepto de concurrencia es fundamental para ambos, y fundamental también para la tecnología de diseño de los sistemas operativos. El Capítulo 5 repasa dos aspectos del control de la concurrencia: la exclusión mutua y la sincronización. La exclusión mutua se refiere a la posibilidad de que múltiples procesos (o hilos) compartan código, recursos, o datos de forma de que sólo uno de ellos tenga acceso al objeto compartido en cada momento. La sincronización se encuentra relacionada con la exclusión mutua: es la posibilidad de que múltiples procesos coordinen sus actividades para intercambiar información. El Capítulo 5 proporciona un amplio tratamiento de los aspectos relativos a la concurrencia, comenzando por un repaso de las consideraciones de diseño implicadas. El capítulo proporciona una revisión del soporte hardware para la concurrencia presentando los mecanismos más importantes para darle soporte: semáforos, monitores, y paso de mensajes.

CAPÍTULO 6. CONCURRENCIA. INTERBLOQUEO E INANICIÓN

El Capítulo 6 muestra dos aspectos más de la concurrencia. Un *interbloqueo* es una situación en la cual dos o más procesos están esperando a que otros miembros del conjunto completen una operación para poder continuar, pero ninguno de los miembros es capaz de hacerlo. Los interbloqueos son un fenómeno difícil de anticipar, y no hay soluciones generales fáciles para éstos. El Capítulo 6 muestra las tres estrategias principales para manejar un interbloqueo: prevenirlo, evitarlo, y detectarlo. La *inanición* se refiere una situación en la cual un proceso se encuentra listo para ejecutar pero se le deniega el acceso al procesador de forma continuada en deferencia a otros procesos. En su mayor parte, la inanición se trata como una cuestión de planificación y por tanto la trataremos en la Parte Cuatro. Aunque el Capítulo 6 se centra en los interbloqueos, la inanición se trata dentro del contexto de las soluciones a los mismos necesarias para evitar el problema de la inanición.

CAPÍTULO 3

Descripción y control de procesos

El diseño de un sistema operativo debe reflejar ciertos requisitos generales. Todos los sistemas operativos multiprogramados, desde los sistemas operativos monousuario como Windows 98 hasta sistemas *mainframes* como IBM z/OS, que son capaces de dar soporte a miles de usuarios, se construyen en torno al concepto de proceso. La mayoría de los requisitos que un sistema operativo debe cumplir se pueden expresar con referencia a los procesos:

- El sistema operativo debe intercalar la ejecución de múltiples procesos, para maximizar la utilización del procesador mientras se proporciona un tiempo de respuesta razonable.
- El sistema operativo debe reservar recursos para los procesos conforme a una política específica (por ejemplo, ciertas funciones o aplicaciones son de mayor prioridad) mientras que al mismo tiempo evita interbloqueos[1].
- Un sistema operativo puede requerir dar soporte a la comunicación entre procesos y la creación de procesos, mediante las cuales ayuda a la estructuración de las aplicaciones.

Se comienza el estudio detallado de los sistemas operativos examinando la forma en la que éstos representan y controlan los procesos. Después de una introducción al concepto de proceso, el capítulo presentará los estados de los procesos, que caracterizan el comportamiento de los mismos. Seguidamente, se presentarán las estructuras de datos que el sistema operativo usa para gestionar los procesos. Éstas incluyen las estructuras para representar el estado de cada proceso así como para registrar características de los mismos que el sistema operativo necesita para alcanzar sus objetivos. Posteriormente, se verá cómo el sistema operativo utiliza estas estructuras para controlar la ejecución de los procesos. Por último, se discute la gestión de procesos en UNIX SVR4. El Capítulo 4 proporciona ejemplos más modernos de gestión de procesos, tales como Solaris, Windows, y Linux.

Nota: en este capítulo hay referencias puntuales a la memoria virtual. La mayoría de las veces podemos ignorar este concepto en relación con los procesos, pero en ciertos puntos de la discusión, las consideraciones sobre memoria virtual se hacen pertinentes. La memoria virtual no se discutirá en detalle hasta el Capítulo 8; ya se ha proporcionado una somera visión general en el Capítulo 2.

3.1. ¿QUÉ ES UN PROCESO?

CONCEPTOS PREVIOS

Antes de definir el término proceso, es útil recapitular algunos de los conceptos ya presentados en los Capítulos 1 y 2:

1. Una plataforma de computación consiste en una colección de recursos hardware, como procesador, memoria, módulos de E/S, relojes, unidades de disco y similares.
2. Las aplicaciones para computadores se desarrollan para realizar determinadas tareas. Suelen aceptar entradas del mundo exterior, realizar algún procesamiento y generar salidas.
3. No es eficiente que las aplicaciones estén escritas directamente para una plataforma hardware específica. Las principales razones son las siguientes:

---

[1] Los interbloqueos se examinarán en el Capítulo 6. Como ejemplo sencillo, un interbloqueo ocurre si dos procesos necesitan dos recursos iguales para continuar y cada uno de los procesos tiene la posesión de uno de los recursos. A menos que se realice alguna acción, cada proceso esperará indefinidamente por conseguir el otro recurso.

a) Numerosas aplicaciones pueden desarrollarse para la misma plataforma, de forma que tiene sentido desarrollar rutinas comunes para acceder a los recursos del computador.

b) El procesador por sí mismo proporciona únicamente soporte muy limitado para la multiprogramación. Es necesario disponer de software para gestionar la compartición del procesador así como otros recursos por parte de múltiples aplicaciones al mismo tiempo.

c) Cuando múltiples aplicaciones están activas al mismo tiempo es necesario proteger los datos, el uso de la E/S y los recursos propios de cada aplicación con respecto a las demás.

4. El sistema operativo se desarrolló para proporcionar una interfaz apropiada para las aplicaciones, rica en funcionalidades, segura y consistente. El sistema operativo es una capa de software entre las aplicaciones y el hardware del computador (Figura 2.1) que da soporte a aplicaciones y utilidades.

5. Se puede considerar que el sistema operativo proporciona una representación uniforme y abstracta de los recursos, que las aplicaciones pueden solicitar y acceder. Los recursos incluyen la memoria principal, las interfaces de red, los sistemas de ficheros, etc. Una vez que el sistema operativo ha creado estas abstracciones de los recursos para que las aplicaciones las usen, debe también controlar su uso. Por ejemplo, un sistema operativo podría permitir compartición y protección de recursos.

Ahora que se conocen los conceptos de aplicaciones, software de sistema y de recursos se está en disposición de hablar sobre cómo un sistema operativo puede, de forma ordenada, gestionar la ejecución de aplicaciones de forma que:

- Los recursos estén disponibles para múltiples aplicaciones.
- El procesador físico se conmute entre múltiples aplicaciones, de forma que todas lleguen a procesarse.
- El procesador y los dispositivos de E/S se puedan usar de forma eficiente.

El enfoque adoptado por todos los sistemas operativos modernos recae en un modelo bajo el cual la ejecución de una aplicación se corresponde con la existencia de uno o más procesos.

PROCESOS Y BLOQUES DE CONTROL DE PROCESOS

Se debe recordar que en el Capítulo 2 se sugirieron diversas definiciones del término *proceso*, incluyendo:

- Un programa en ejecución.
- Una instancia de un programa ejecutado en un computador.
- La entidad que se puede asignar y ejecutar en un procesador.
- Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados.

También se puede pensar en un proceso como en una entidad que consiste en un número de elementos. Los dos elementos esenciales serían el **código de programa** (que puede compartirse con otros procesos que estén ejecutando el mismo programa) y un **conjunto de datos** asociados a dicho

código. Supongamos que el procesador comienza a ejecutar este código de programa, y que nos referiremos a esta entidad en ejecución como un proceso. En cualquier instante puntual del tiempo, *mientras el proceso está en ejecución*, este proceso se puede caracterizar por una serie de elementos, incluyendo los siguientes:

- **Identificador.** Un identificador único asociado a este proceso, para distinguirlo del resto de procesos.
- **Estado.** Si el proceso está actualmente corriendo, está en el estado *en ejecución*.
- **Prioridad**: Nivel de prioridad relativo al resto de procesos.
- **Contador de programa.** La dirección de la siguiente instrucción del programa que se ejecutará.
- **Punteros a memoria.** Incluye los punteros al código de programa y los datos asociados a dicho proceso, además de cualquier bloque de memoria compartido con otros procesos.
- **Datos de contexto.** Estos son datos que están presenten en los registros del procesador cuando el proceso está corriendo.
- **Información de estado de E/S.** Incluye las peticiones de E/S pendientes, dispositivos de E/S (por ejemplo, una unidad de cinta) asignados a dicho proceso, una lista de los ficheros en uso por el mismo, etc.
- **Información de auditoría.** Puede incluir la cantidad de tiempo de procesador y de tiempo de reloj utilizados, así como los límites de tiempo, registros contables, etc.

La información de la lista anterior se almacena en una estructura de datos, que se suele llamar bloque de control de proceso (*process control block*) (Figura 3.1), que el sistema operativo crea y gestiona. El punto más significativo en relación al bloque de control de proceso, o BCP, es que contiene suficiente información de forma que es posible interrumpir el proceso cuando está corriendo y posteriormente restaurar su estado de ejecución como si no hubiera habido interrupción alguna. El BCP es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y proporcionar multiprogramación. Cuando un proceso se interrumpe, los valores actuales del contador de programa y los registros del procesador (datos de contexto) se guardan en los campos correspondientes del BCP y el estado del proceso se cambia a cualquier otro valor, como *bloqueado* o *listo* (descritos a continuación). El sistema operativo es libre ahora para poner otro proceso en estado de ejecución. El contador de programa y los datos de contexto se recuperan y cargan en los registros del procesador y este proceso comienza a correr.

De esta forma, se puede decir que un proceso está compuesto del código de programa y los datos asociados, además del bloque de control de proceso o BCP. Para un computador monoprocesador, en un instante determinado, como máximo un único proceso puede estar corriendo y dicho proceso estará en el estado *en ejecución*.

3.2. ESTADOS DE LOS PROCESOS

Como se acaba de comentar, para que un programa se ejecute, se debe crear un proceso o tarea para dicho programa. Desde el punto de vista del procesador, él ejecuta instrucciones de su repertorio de instrucciones en una secuencia dictada por el cambio de los valores del registro contador de programa. A lo largo del tiempo, el contador de programa puede apuntar al código de diferentes programas que son parte de diferentes procesos. Desde el punto de vista de un programa individual, su ejecución implica una secuencia de instrucciones dentro de dicho programa.

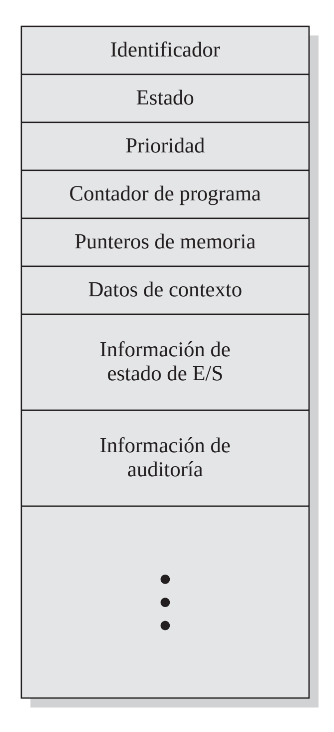


Figura 3.1. Bloque de control de programa (BCP) simplificado.

Se puede caracterizar el comportamiento de un determinado proceso, listando la secuencia de instrucciones que se ejecutan para dicho proceso. A esta lista se la denomina **traza** del proceso. Se puede caracterizar el comportamiento de un procesador mostrando cómo las trazas de varios procesos se entrelazan.

Considere un ejemplo. La Figura 3.2 muestra el despliegue en memoria de tres procesos. Para simplificar la exposición, se asume que dichos procesos no usan memoria virtual; por tanto, los tres procesos están representados por programas que residen en memoria principal. De manera adicional, existe un pequeño programa **activador** (*dispatcher*) que intercambia el procesador de un proceso a otro. La Figura 3.3 muestra las trazas de cada uno de los procesos en los primeros instantes de ejecución. Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C. El proceso B ejecuta 4 instrucciones y se asume que la cuarta instrucción invoca una operación de E/S, a la cual el proceso debe esperar.

Ahora vea estas trazas desde el punto de vista del procesador. La Figura 3.4 muestra las trazas entrelazadas resultante de los 52 primeros ciclos de ejecución (por conveniencia los ciclos de instrucciones han sido numerados). En este ejemplo, se asume que el sistema operativo sólo deja que un proceso continúe durante seis ciclos de instrucción, después de los cuales se interrumpe; lo cual previene que un solo proceso monopolice el uso del tiempo del procesador. Como muestra la Figura 3.4, las primeras seis instrucciones del proceso A se ejecutan seguidas de una alarma de temporización (*time-out*) y de la ejecución de cierto código del activador, que ejecuta seis instrucciones antes de devolver el control al proceso B[2]. Después de que se ejecuten cuatro instrucciones, el proceso B solicita una acción de

---

[2] El reducido número de instrucciones ejecutadas por los procesos y por el planificador es irreal; se ha utilizado para simplificar el ejemplo y clarificar las explicaciones.

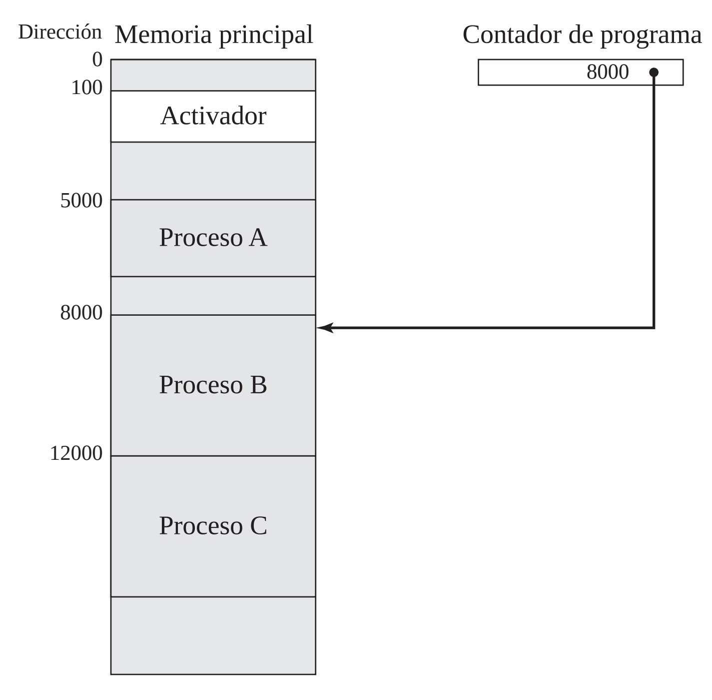


Figura 3.2. Instantánea de un ejemplo de ejecución (Figura 3.4) en el ciclo de instrucción 13.

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011
(a) Traza del Proceso A	**(b) Traza del Proceso B**	**(c) Traza del Proceso C**

5000 = Dirección de comienzo del programa del Proceso A.
8000 = Dirección de comienzo del programa del Proceso B.
12000 = Dirección de comienzo del programa del Proceso C.

Figura 3.3. Traza de los procesos de la Figura 3.2.

E/S, para la cual debe esperar. Por tanto, el procesador deja de ejecutar el proceso B y pasa a ejecutar el proceso C, por medio del activador. Después de otra alarma de temporización, el procesador vuelve al proceso A. Cuando este proceso llega a su temporización, el proceso B aún estará esperando que se complete su operación de E/S, por lo que el activador pasa de nuevo al proceso C.

UN MODELO DE PROCESO DE DOS ESTADOS

La responsabilidad principal del sistema operativo es controlar la ejecución de los procesos; esto incluye determinar el patrón de entrelazado para la ejecución y asignar recursos a los procesos. El primer paso en el diseño de un sistema operativo para el control de procesos es describir el comportamiento que se desea que tengan los procesos.

Se puede construir el modelo más simple posible observando que, en un instante dado, un proceso está siendo ejecutando por el procesador o no. En este modelo, un proceso puede estar en dos estados: Ejecutando o No Ejecutando, como se muestra en la Figura 3.5a. Cuando el sistema operativo crea un nuevo proceso, crea el bloque de control de proceso (BCP) para el nuevo proceso e inserta dicho proceso en el sistema en estado No Ejecutando. El proceso existe, es conocido por el sistema operativo, y está esperando su oportunidad de ejecutar. De cuando en cuando, el proceso actualmente en ejecución se interrumpirá y una parte del sistema operativo, el activador, seleccionará otro proceso

1	5000	
2	5001	
3	5002	
4	5003	
5	5004	
6	5005	
		Temporización
7	100	
8	101	
9	102	
10	103	
11	104	
12	105	
13	8000	
14	8001	
15	8002	
16	8003	
		Petición de E/S
17	100	
18	101	
19	102	
20	103	
21	104	
22	105	
23	12000	
24	12001	
25	12002	
26	12003	
27	12004	
28	12005	
		Temporización
29	100	
30	101	
31	102	
32	103	
33	104	
34	105	
35	5006	
36	5007	
37	5008	
38	5009	
39	5010	
40	5011	
		Temporización
41	100	
42	101	
43	102	
44	103	
45	104	
46	105	
47	12006	
48	12007	
49	12008	
50	12009	
51	12010	
52	12011	
		Temporización

100 = Dirección de comienzo del programa activador.
Las zonas sombreadas indican la ejecución del proceso de activación;
la primera y la tercera columna cuentan ciclos de instrucciones;
la segunda y la cuarta columna las direcciones de las instrucciones que se ejecutan

Figura 3.4. Traza combinada de los procesos de la Figura 3.2.

a ejecutar. El proceso saliente pasará del estado Ejecutando a No Ejecutando y pasará a Ejecutando un nuevo proceso.

De este modelo simple, ya se puede apreciar algo del diseño de los elementos del sistema operativo. Cada proceso debe representarse de tal manera que el sistema operativo pueda seguirle la pista. Es decir, debe haber información correspondiente a cada proceso, incluyendo el estado actual y su localización en memoria; esto es el bloque de control de programa. Los procesos que no están ejecutando deben estar en una especie de cola, esperando su turno de ejecución. La Figura 3.5b sugiere esta estructura. Existe una sola cola cuyas entradas son punteros al BCP de un proceso en particular. Alternativamente, la cola debe consistir en una lista enlazada de bloques de datos, en la cual cada bloque representa un proceso; exploraremos posteriormente esta última implementación.

Podemos describir el comportamiento del activador en términos de este diagrama de colas. Un proceso que se interrumpe se transfiere a la cola de procesos en espera. Alternativamente, si el proceso ha finalizado o ha sido abortado, se descarta (sale del sistema). En cualquier caso, el activador selecciona un proceso de la cola para ejecutar.

CREACIÓN Y TERMINACIÓN DE PROCESOS

Antes de intentar refinar nuestro sencillo modelo de dos estados, resultará útil hablar de la creación y terminación de los procesos; por último, y de forma independiente del modelo de comportamiento de procesos que se use, la vida de un proceso está acotada entre su creación y su terminación.

Creación de un proceso. Cuando se va a añadir un nuevo proceso a aquellos que se están gestionando en un determinado momento, el sistema operativo construye las estructuras de datos que se usan para manejar el proceso (como se describió en la Sección 3.3) y reserva el espacio de direcciones en memoria principal para el proceso. Estas acciones constituyen la creación de un nuevo proceso.

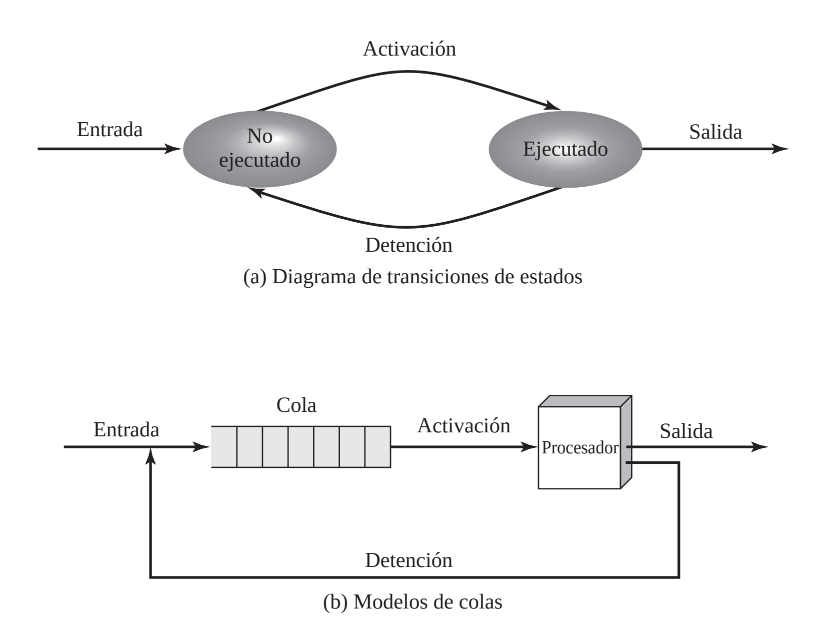


Figura 3.5. Modelo de proceso de dos estados.

Existen cuatro eventos comunes que llevan a la creación de un proceso, como se indica en la Tabla 3.1. En un entorno por lotes, un proceso se crea como respuesta a una solicitud de trabajo. En un entorno interactivo, un proceso se crea cuando un nuevo usuario entra en el sistema. En ambos casos el sistema operativo es responsable de la creación de nuevos procesos. Un sistema operativo puede, a petición de una aplicación, crear procesos. Por ejemplo, si un usuario solicita que se imprima un fichero, el sistema operativo puede crear un proceso que gestione la impresión. El proceso solicitado puede, de esta manera, operar independientemente del tiempo requerido para completar la tarea de impresión.

TABLA 3.1. Razones para la creación de un proceso.

Nuevo proceso de lotes	El sistema operativo dispone de un flujo de control de lotes de trabajos, habitualmente una cinta un disco. Cuando el sistema operativo está listo para procesar un nuevo trabajo, leerá la siguiente secuencia de mandatos de control de trabajos.
Sesión interactiva	Un usuario desde un terminal entra en el sistema.
Creado por el sistema operativo para proporcionar un servicio	El sistema operativo puede crear un proceso para realizar una función en representación de un programa de usuario, sin que el usuario tenga que esperar (por ejemplo, un proceso para controlar la impresión).
Creado por un proceso existente	Por motivos de modularidad o para explotar el paralelismo, un programa de usuario puede ordenar la creación de un número de procesos.

Tradicionalmente, un sistema operativo creaba todos los procesos de una forma que era transparente para usuarios y programas. Esto aún es bastante común en muchos sistemas operativos contemporáneos. Sin embargo, puede ser muy útil permitir a un proceso la creación de otro. Por ejemplo, un proceso de aplicación puede generar otro proceso para recibir los datos que la aplicación está generando y para organizar esos datos de una forma apropiada para su posterior análisis. El nuevo proceso ejecuta en paralelo con el proceso original y se activa cuando los nuevos datos están disponibles. Esta organización puede ser muy útil en la estructuración de la aplicación. Otro ejemplo, un proceso servidor (por ejemplo, un servidor de impresoras, servidor de ficheros) puede generar un proceso por cada solicitud que esté manejando. Cuando un sistema operativo crea un proceso a petición explícita de otro proceso, dicha acción se denomina ***creación del proceso***.

Cuando un proceso lanza otro, al primero se le denomina **proceso padre**, y al proceso creado se le denomina **proceso hijo**. Habitualmente, la relación entre procesos necesita comunicación y cooperación entre ellos. Alcanzar esta cooperación es una tarea complicada para un programador; este aspecto se verá en el Capítulo 5.

Terminación de procesos. La Tabla 3.2 resume las razones típicas para la terminación de un proceso. Todo sistema debe proporcionar los mecanismos mediante los cuales un proceso indica su finalización, o que ha completado su tarea. Un trabajo por lotes debe incluir una instrucción HALT o una llamada a un servicio de sistema operativo especifica para su terminación. En el caso anterior, la instrucción HALT generará una interrupción para indicar al sistema operativo que dicho proceso ha finalizado. Para una aplicación interactiva, las acciones del usuario indicarán cuando el proceso ha terminado. Por ejemplo, en un sistema de tiempo compartido, el proceso de un usuario en particular puede terminar cuando el usuario sale del sistema o apaga su terminal. En un ordenador personal o una estación de trabajo, el usuario puede salir de una aplicación (por ejemplo, un procesador de texto o una

hoja de cálculo). Todas estas acciones tienen como resultado final la solicitud de un servicio al sistema operativo para terminar con el proceso solicitante.

Adicionalmente, un número de error o una condición de fallo puede llevar a la finalización de un proceso. La Tabla 3.2 muestra una lista de las condiciones más habituales de finalización por esos motivos[3].

Tabla 3.2. Razones para la terminación de un proceso.

Finalización normal	El proceso ejecuta una llamada al sistema operativo para indicar que ha completado su ejecución
Límite de tiempo excedido	El proceso ha ejecutado más tiempo del especificado en un límite máximo. Existen varias posibilidades para medir dicho tiempo. Estas incluyen el tiempo total utilizado, el tiempo utilizado únicamente en ejecución, y, en el caso de procesos interactivos, la cantidad de tiempo desde que el usuario realizó la última entrada.
Memoria no disponible	El proceso requiere más memoria de la que el sistema puede proporcionar.
Violaciones de frontera	El proceso trata de acceder a una posición de memoria a la cual no tiene acceso permitido.
Error de protección	El proceso trata de usar un recurso, por ejemplo un fichero, al que no tiene permitido acceder, o trata de utilizarlo de una forma no apropia, por ejemplo, escribiendo en un fichero de sólo lectura.
Error aritmético	El proceso trata de realizar una operación de cálculo no permitida, tal como una división por 0, o trata de almacenar números mayores de los que la representación hardware puede codificar.
Límite de tiempo	El proceso ha esperado más tiempo que el especificado en un valor máximo para que se cumpla un determinado evento.
Fallo de E/S	Se ha producido un error durante una operación de entrada o salida, por ejemplo la imposibilidad de encontrar un fichero, fallo en la lectura o escritura después de un límite máximo de intentos (cuando, por ejemplo, se encuentra un área defectuosa en una cinta), o una operación inválida (la lectura de una impresora en línea).
Instrucción no válida	El proceso intenta ejecutar una instrucción inexistente (habitualmente el resultado de un salto a un área de datos y el intento de ejecutar dichos datos).
Instrucción privilegiada	El proceso intenta utilizar una instrucción reservada al sistema operativo.
Uso inapropiado de datos	Una porción de datos es de tipo erróneo o no se encuentra inicializada.
Intervención del operador por el sistema operativo	Por alguna razón, el operador o el sistema operativo ha finalizado el proceso (por ejemplo, se ha dado una condición de interbloqueo).
Terminación del proceso padre	Cuando un proceso padre termina, el sistema operativo puede automáticamente finalizar todos los procesos hijos descendientes de dicho padre.
Solicitud del proceso padre	Un proceso padre habitualmente tiene autoridad para finalizar sus propios procesos descendientes.

---

[3] Un sistema operativo compasivo podría en algunos casos permitir al usuario recuperarse de un fallo sin terminar el proceso. Por ejemplo, si un usuario solicita acceder a un fichero y se deniega ese acceso, el sistema operativo podría simplemente informar al usuario de ese hecho y permitir que el proceso continúe.

Por último, en ciertos sistemas operativos, un proceso puede terminarse por parte del proceso que lo creó o cuando dicho proceso padre a su vez ha terminado.

MODELO DE PROCESO DE CINCO ESTADOS

Si todos los procesos estuviesen siempre preparados para ejecutar, la gestión de colas proporcionada en la Figura 3.5b sería efectiva. La cola es una lista de tipo FIFO y el procesador opera siguiendo una estrategia cíclica (*round-robin* o turno rotatorio) sobre todos los procesos disponibles (cada proceso de la cola tiene cierta cantidad de tiempo, por turnos, para ejecutar y regresar de nuevo a la cola, a menos que se bloquee). Sin embargo, hasta con el sencillo ejemplo que vimos antes, esta implementación es inadecuada: algunos procesos que están en el estado de No Ejecutando están listos para ejecutar, mientras que otros están bloqueados, esperando a que se complete una operación de E/S. Por tanto, utilizando una única cola, el activador no puede seleccionar únicamente los procesos que lleven más tiempo en la cola. En su lugar, debería recorrer la lista buscando los procesos que no estén bloqueados y que lleven en la cola más tiempo.

Una forma más natural para manejar esta situación es dividir el estado de No Ejecutando en dos estados, Listo y Bloqueado. Esto se muestra la Figura 3.6. Para gestionarlo correctamente, se han añadido dos estados adicionales que resultarán muy útiles. Estos cinco estados en el nuevo diagrama son los siguientes:

- **Ejecutando.** El proceso está actualmente en ejecución. Para este capítulo asumimos que el computador tiene un único procesador, de forma que sólo un proceso puede estar en este estado en un instante determinado.
- **Listo.** Un proceso que se prepara para ejecutar cuando tenga oportunidad.
- **Bloqueado.** Un proceso que no puede ejecutar hasta que se cumpla un evento determinado o se complete una operación E/S.
- **Nuevo.** Un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. Típicamente, se trata de un nuevo proceso que no ha sido cargado en memoria principal, aunque su bloque de control de proceso (BCP) si ha sido creado.
- **Saliente.** Un proceso que ha sido liberado del grupo de procesos ejecutables por el sistema operativo, debido a que ha sido detenido o que ha sido abortado por alguna razón.

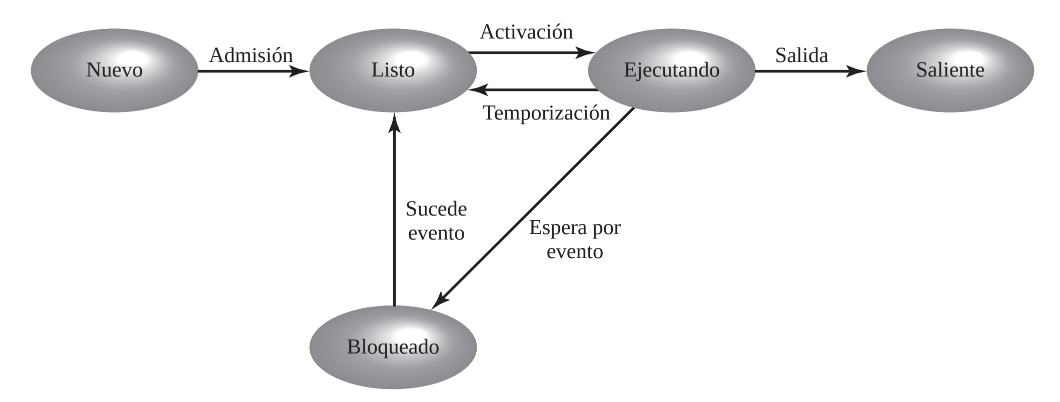


Figura 3.6. Modelo de proceso de cinco estados.

Los estados Nuevo y Saliente son útiles para construir la gestión de procesos. El estado Nuevo se corresponde con un proceso que acaba de ser definido. Por ejemplo, si un nuevo usuario intenta entrar dentro de un sistema de tiempo compartido o cuando se solicita un nuevo trabajo a un sistema de proceso por lotes, el sistema operativo puede definir un nuevo proceso en dos etapas. Primero, el sistema operativo realiza todas las tareas internas que correspondan. Se asocia un identificador a dicho proceso. Se reservan y construyen todas aquellas tablas que se necesiten para gestionar al proceso. En este punto, el proceso se encuentra el estado Nuevo. Esto significa que el sistema operativo ha realizado todas las tareas necesarias para crear el proceso pero el proceso en sí, aún no se ha puesto en ejecución. Por ejemplo, un sistema operativo puede limitar el número de procesos que puede haber en el sistema por razones de rendimiento o limitaciones de memoria principal. Mientras un proceso está en el estado Nuevo, la información relativa al proceso que se necesite por parte del sistema operativo se mantiene en tablas de control de memoria principal. Sin embargo, el proceso en sí mismo no se encuentra en memoria principal. Esto es, el código de programa a ejecutar no se encuentra en memoria principal, y no se ha reservado ningún espacio para los datos asociados al programa. Cuando un proceso se encuentra en el estado Nuevo, el programa permanece en almacenamiento secundario, normalmente en disco[4].

De forma similar, un proceso sale del sistema en dos fases. Primero, el proceso termina cuando alcanza su punto de finalización natural, cuando es abortado debido a un error no recuperable, o cuando otro proceso con autoridad apropiada causa que el proceso se aborte. La terminación mueve el proceso al estado Saliente. En este punto, el proceso no es elegible de nuevo para su ejecución. Las tablas y otra información asociada con el trabajo se encuentran temporalmente preservadas por el sistema operativo, el cual proporciona tiempo para que programas auxiliares o de soporte extraigan la información necesaria. Por ejemplo, un programa de auditoría puede requerir registrar el tiempo de proceso y otros recursos utilizados por este proceso saliente con objeto de realizar una contabilidad de los recursos del sistema. Un programa de utilidad puede requerir extraer información sobre el histórico de los procesos por temas relativos con el rendimiento o análisis de la utilización. Una vez que estos programas han extraído la información necesaria, el sistema operativo no necesita mantener ningún dato relativo al proceso y el proceso se borra del sistema.

La Figura 3.6 indica que tipos de eventos llevan a cada transición de estado para cada proceso; las posibles transiciones son las siguientes:

- **Null → Nuevo.** Se crea un nuevo proceso para ejecutar un programa. Este evento ocurre por cualquiera de las relaciones indicadas en la tabla 3.1.
- **Nuevo → Listo.** El sistema operativo mueve a un proceso del estado nuevo al estado listo cuando éste se encuentre preparado para ejecutar un nuevo proceso. La mayoría de sistemas fijan un límite basado en el número de procesos existentes o la cantidad de memoria virtual que se podrá utilizar por parte de los procesos existentes. Este límite asegura que no haya demasiados procesos activos y que se degrade el rendimiento sistema.
- **Listo → Ejecutando.** Cuando llega el momento de seleccionar un nuevo proceso para ejecutar, el sistema operativo selecciona uno los procesos que se encuentre en el estado Listo. Esta es una tarea la lleva acabo el planificador. El planificador se estudiará más adelante en la Parte Cuatro.
- **Ejecutando → Saliente.** el proceso actual en ejecución se finaliza por parte del sistema operativo tanto si el proceso indica que ha completado su ejecución como si éste se aborta. *Véase* tabla 3.2.

---

[4] En las explicaciones de este párrafo, hemos ignorado el concepto de memoria virtual. En sistemas que soporten la memoria virtual, cuando un proceso se mueve de Nuevo a Listo, su código de programa y sus datos se cargan en memoria virtual. La memoria virtual se ha explicado brevemente en el Capítulo 2 y se examinará con más detalle en el Capítulo 8.

- **Ejecutando → Listo.** La razón más habitual para esta transición es que el proceso en ejecución haya alcanzado el máximo tiempo posible de ejecución de forma ininterrumpida; prácticamente todos los sistemas operativos multiprogramados imponen este tipo de restricción de tiempo. Existen otras posibles causas alternativas para esta transición, que no están incluidas en todos los sistemas operativos. Es de particular importancia el caso en el cual el sistema operativo asigna diferentes niveles de prioridad a diferentes procesos. Supóngase, por ejemplo, que el proceso A está ejecutando a un determinado nivel de prioridad, y el proceso B, a un nivel de prioridad mayor, y que se encuentra bloqueado. Si el sistema operativo se da cuenta de que se produce un evento al cual el proceso B está esperando, moverá el proceso B al estado de Listo. Esto puede interrumpir al proceso A y poner en ejecución al proceso B. Decimos, en este caso, que el sistema operativo ha expulsado al proceso A[5]. Adicionalmente, un proceso puede voluntariamente dejar de utilizar el procesador. Un ejemplo son los procesos que realiza alguna función de auditoría o de mantenimiento de forma periódica.
- **Ejecutando → Bloqueado.** Un proceso se pone en el estado Bloqueado si solicita algo por lo cual debe esperar. Una solicitud al sistema operativo se realiza habitualmente por medio de una llamada al sistema; esto es, una llamada del proceso en ejecución a un procedimiento que es parte del código del sistema operativo. Por ejemplo, un proceso ha solicitado un servicio que el sistema operativo no puede realizar en ese momento. Puede solicitar un recurso, como por ejemplo un fichero o una sección compartida de memoria virtual, que no está inmediatamente disponible. Cuando un proceso quiere iniciar una acción, tal como una operación de E/S, que debe completarse antes de que el proceso continúe. Cuando un proceso se comunica con otro, un proceso puede bloquearse mientras está esperando a que otro proceso le proporcione datos o esperando un mensaje de ese otro proceso.
- **Bloqueado → Listo.** Un proceso en estado Bloqueado se mueve al estado Listo cuando sucede el evento por el cual estaba esperando.
- **Listo → Saliente.** Por claridad, esta transición no se muestra en el diagrama de estados. En algunos sistemas, un padre puede terminar la ejecución de un proceso hijo en cualquier momento. También, si el padre termina, todos los procesos hijos asociados con dicho padre pueden finalizarse.
- **Bloqueado → Saliente.** Se aplican los comentarios indicados en el caso anterior.

Si regresamos a nuestro sencillo ejemplo, Figura 3.7, ahí se muestra la transición entre cada uno de los estados de proceso. La figura 3.8a sugiere la forma de aplicar un esquema de dos colas: la colas de Listos y la cola de Bloqueados. Cada proceso admitido por el sistema, se coloca en la cola de Listos. Cuando llega el momento de que el sistema operativo seleccione otro proceso a ejecutar, selecciona uno de la cola de Listos. En ausencia de un esquema de prioridad, esta cola puede ser una lista de tipo FIFO (*first-in-first-out*). Cuando el proceso en ejecución termina de utilizar el procesador, o bien finaliza o bien se coloca en la cola de Listos o de Bloqueados, dependiendo de las circunstancias. Por último, cuando sucede un evento, cualquier proceso en la cola de Bloqueados que únicamente esté esperando a dicho evento, se mueve a la cola de Listos.

Esta última transición significa que, cuando sucede un evento, el sistema operativo debe recorrer la cola entera de Bloqueados, buscando aquellos procesos que estén esperando por dicho evento. En

---

[5] En general, el término **expulsión** (*preemption*) se define como la reclamación de un recurso por parte de un proceso antes de que el proceso que lo poseía finalice su uso. En este caso, el recurso es el procesador. El proceso está ejecutando y puede continuar su ejecución pero es expulsado por otro proceso que va a entrar a ejecutar.

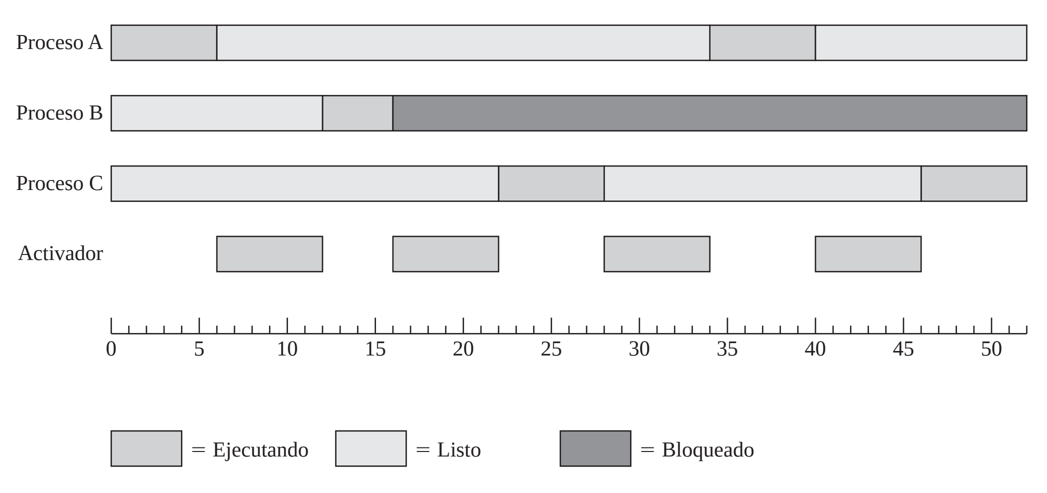


Figura 3.7. Estado de los procesos de la traza de la Figura 3.4.

los sistemas operativos con muchos procesos, esto puede significar cientos o incluso miles de procesos en esta lista, por lo que sería mucho más eficiente tener una cola por cada evento. De esta forma, cuando sucede un evento, la lista entera de procesos de la cola correspondiente se movería al estado de Listo (Figura 3. 8b).

Un refinamiento final sería: si la activación de procesos está dictada por un esquema de prioridades, sería conveniente tener varias colas de procesos listos, una por cada nivel de prioridad. El sistema operativo podría determinar cual es el proceso listo de mayor prioridad simplemente seleccionando éstas en orden.

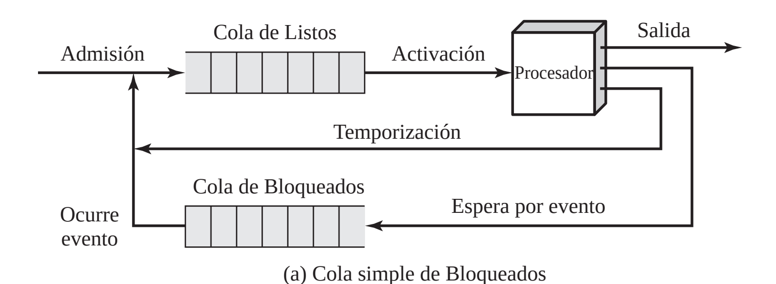
PROCESOS SUSPENDIDOS

La necesidad de intercambio o *swapping*. Los tres principales estados descritos (Listo, Ejecutando, Bloqueado) proporcionan una forma sistemática de modelizar el comportamiento de los procesos y diseñar la implementación del sistema operativo. Se han construido algunos sistemas operativos utilizando únicamente estos tres estados.

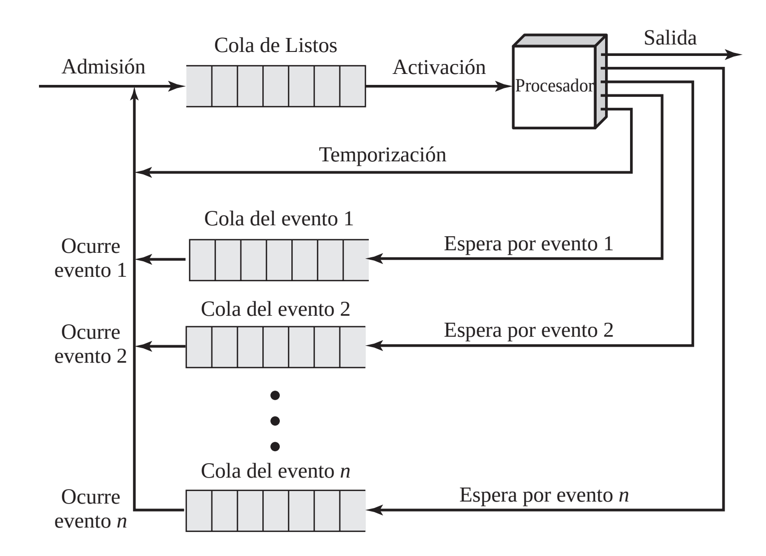
Sin embargo, existe una buena justificación para añadir otros estados al modelo. Para ver este beneficio de nuevos estados, vamos a suponer un sistema que no utiliza memoria virtual. Cada proceso que se ejecuta debe cargarse completamente en memoria principal. Por ejemplo, en la Figura 3.8b, todos los procesos en todas las colas deben residir en memoria principal.

Recuérdese que la razón de toda esta compleja maquinaria es que las operaciones de E/S son mucho más lentas que los procesos de cómputo y, por tanto, el procesador en un sistema monoprogramado estaría ocioso la mayor parte del tiempo. Pero los ajustes de la Figura 3.8b no resuelven completamente el problema. Es verdad que, en este caso, la memoria almacena múltiples procesos y el procesador puede asignarse a otro proceso si el que lo usa se queda bloqueado. La diferencia de velocidad entre el procesador y la E/S es tal que sería muy habitual que todos los procesos en memoria se encontrasen a esperas de dichas operaciones. Por tanto, incluso en un sistema multiprogramado, el procesador puede estar ocioso la mayor parte del tiempo.

¿Qué se puede hacer? La memoria principal puede expandirse para acomodar más procesos. Hay dos fallos en esta solución. Primero, existe un coste asociado a la memoria principal, que, desde un



(a) Cola simple de Bloqueados



(b) Múltiples colas de Bloqueados

Figura 3.8. Modelo de colas de la Figura 3.6.

coste reducido a nivel de bytes, comienza a incrementarse según nos acercamos a un almacenamiento de gigabytes. Segundo, el apetito de los programas a nivel de memoria ha crecido tan rápido como ha bajado el coste de las memorias. De forma que las grandes memorias actuales han llevado a ejecutar procesos de gran tamaño, no más procesos.

Otra solución es el *swapping* (memoria de intercambio), que implica mover parte o todo el proceso de memoria principal al disco. Cuando ninguno de los procesos en memoria principal se encuentra en estado Listo, el sistema operativo intercambia uno de los procesos bloqueados a disco, en la cola de Suspendidos. Esta es una lista de procesos existentes que han sido temporalmente expulsados de la memoria principal, o suspendidos. El sistema operativo trae otro proceso de la cola de Suspendidos o responde a una solicitud de un nuevo proceso. La ejecución continúa con los nuevos procesos que han llegado.

El *swapping*, sin embargo, es una operación de E/S, y por tanto existe el riesgo potencial de hacer que el problema empeore. Pero debido a que la E/S en disco es habitualmente más rápida que la E/S sobre otros sistemas (por ejemplo, comparado con cinta o impresora), el *swapping* habitualmente mejora el rendimiento del sistema.

Con el uso de *swapping* tal y como se ha escrito, debe añadirse un nuevo estado a nuestro modelo de comportamiento de procesos (Figura 3.9a): el estado Suspendido. Cuando todos los procesos en

memoria principal se encuentran en estado Bloqueado, el sistema operativo puede suspender un proceso poniéndolo en el estado Suspendido y transfiriéndolo a disco. El espacio que se libera en memoria principal puede usarse para traer a otro proceso.

Cuando el sistema operativo ha realizado la operación de *swap* (transferencia a disco de un proceso), tiene dos opciones para seleccionar un nuevo proceso para traerlo a memoria principal: puede admitir un nuevo proceso que se haya creado o puede traer un proceso que anteriormente se encontrase en estado de Suspendido. Parece que sería preferible traer un proceso que anteriormente estuviese suspendido, para proporcionar dicho servicio en lugar de incrementar la carga total del sistema.

Pero, esta línea de razonamiento presenta una dificultad: todos los procesos que fueron suspendidos se encontraban previamente en el estado de Bloqueado en el momento de su suspensión. Claramente no sería bueno traer un proceso bloqueado de nuevo a memoria porque podría no encontrarse todavía listo para la ejecución. Se debe reconocer, sin embargo, que todos los procesos en estado Suspendido estaban originalmente en estado Bloqueado, en espera de un evento en particular. Cuando el evento sucede, el proceso no está Bloqueado y está potencialmente disponible para su ejecución.

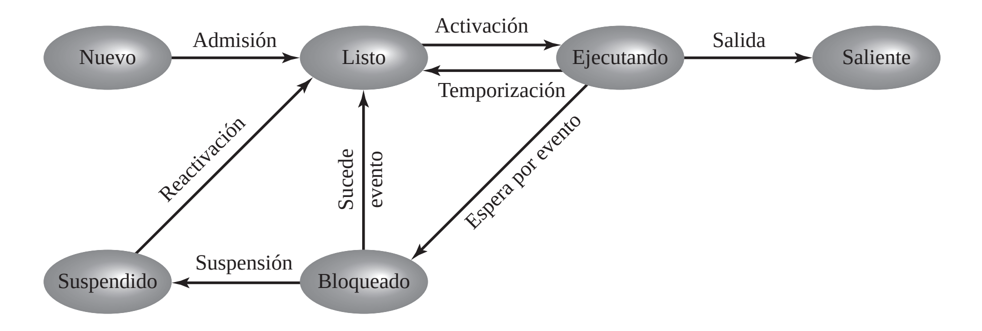
De esta forma, necesitamos replantear este aspecto del diseño. Hay dos conceptos independientes aquí: si un proceso está esperando a un evento (Bloqueado o no) y si un proceso está transferido de memoria a disco (suspendido o no). Para describir estas combinaciones de 2×2 necesitamos cuatro estados:

- **Listo.** El proceso está en memoria principal disponible para su ejecución.
- **Bloqueado.** El proceso está en memoria principal y esperando un evento.
- **Bloqueado/Suspendido.** El proceso está en almacenamiento secundario y esperando un evento.
- **Listo/Suspendido.** El proceso está en almacenamiento secundario pero está disponible para su ejecución tan pronto como sea cargado en memoria principal.

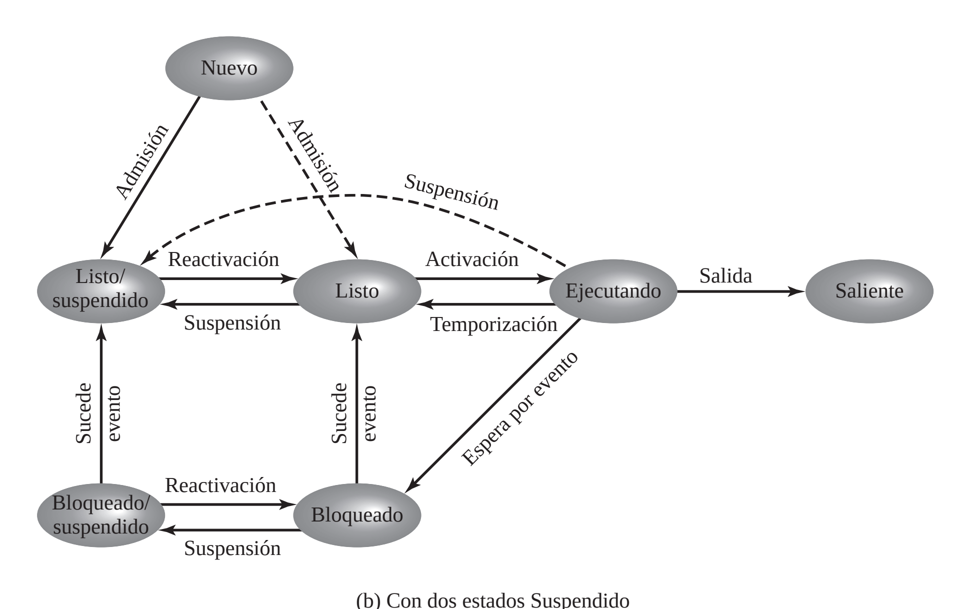
Antes de ir a ver un diagrama de transiciones de estado que incluya estos dos nuevos estados Suspendidos, se debe mencionar otro punto. La explicación ha asumido que no se utiliza memoria virtual y que un proceso está completamente en memoria principal, o completamente fuera de la misma. Con un esquema de memoria virtual, es posible ejecutar un proceso que está sólo parcialmente en memoria principal. Si se hace referencia a una dirección de proceso que no se encuentra en memoria principal, la porción correspondiente del proceso se trae a ella. El uso de la memoria principal podría parecer que elimina la necesidad del *swapping* explícito, porque cualquier dirección necesitada por un proceso que la demande puede traerse o transferirse de disco a memoria por el propio hardware de gestión de memoria del procesador. Sin embargo, como veremos en el Capítulo 8 el rendimiento de los sistemas de memoria virtual pueden colapsarse si hay un número suficientemente grande de procesos activos, todos ellos están parcialmente en memoria principal. De esta forma, incluso con un sistema de memoria virtual, el sistema operativo necesita hacer *swapping* de los procesos de forma explícita y completa, de vez en cuando, con la intención de mejorar el rendimiento.

Veamos ahora, en la Figura 3.9b, el modelo de transición de estados que ha sido diseñado. (Las líneas punteadas de la figura indican transiciones posibles pero no necesarias.) Las nuevas transiciones más importantes son las siguientes:

- **Bloqueado → Bloqueado/Suspendido.** Si no hay procesos listos, entonces al menos uno de los procesos bloqueados se transfiere al disco para hacer espacio para otro proceso que no se encuentra bloqueado. Esta transición puede realizarse incluso si hay procesos listos disponi-



(a) Con un único estado Suspendido



(b) Con dos estados Suspendido

Figura 3.9. Diagrama de transición de estados de procesos con estado suspendidos.

bles, si el sistema operativo determina que el proceso actualmente en ejecución o los procesos listos que desea ejecutar requieren más memoria principal para mantener un rendimiento adecuado.

- **Bloqueado/Suspendido → Listo/Suspendido.** Un proceso en el estado Bloqueado/Suspendido se mueve al estado Listo/Suspendido cuando sucede un evento al que estaba esperando. Nótese que esto requiere que la información de estado concerniente a un proceso suspendido sea accesible para el sistema operativo.
- **Listo/Suspendido → Listo.** Cuando no hay más procesos listos en memoria principal, el sistema operativo necesitará traer uno para continuar la ejecución. Adicionalmente, puede darse el caso de que un proceso en estado Listo/Suspendido tenga mayor prioridad que cualquiera de los procesos en estado Listo. En este caso, el sistema operativo puede haberse diseñado para determinar que es más importante traer un proceso de mayor prioridad que para minimizar el efecto del *swapping*.

- **Listo → Listo/Suspendido.** Normalmente, el sistema operativo preferirá suspender procesos bloqueados que un proceso Listo, porque un proceso Listo se puede ejecutar en ese momento, mientras que un proceso Bloqueado ocupa espacio de memoria y no se puede ejecutar. Sin embargo, puede ser necesario suspender un proceso Listo si con ello se consigue liberar un bloque suficientemente grande de memoria. También el sistema operativo puede decidir suspender un proceso Listo de baja prioridad antes que un proceso Bloqueado de alta prioridad, si se cree que el proceso Bloqueado estará pronto listo.

Otras transiciones interesantes de considerar son las siguientes:

- **Nuevo → Listo/Suspendido y Nuevo a Listo.** Cuando se crea un proceso nuevo, puede añadirse a la cola de Listos o a la cola de Listos/Suspendidos. En cualquier caso, el sistema operativo puede crear un bloque de control de proceso (BCP) y reservar el espacio de direcciones del proceso. Puede ser preferible que el sistema operativo realice estas tareas internas cuanto antes, de forma que pueda disponer de suficiente cantidad de procesos no bloqueados. Sin embargo, con esta estrategia, puede ocurrir que no haya espacio suficiente en memoria principal para el nuevo proceso; de ahí el uso de la transición (Nuevo→Listo/Suspendido). Por otro lado, deseamos hacer hincapié en que la filosofía de creación de procesos diferida, haciéndolo cuanto más tarde, hace posible reducir la sobrecarga del sistema operativo y le permite realizar las tareas de creación de procesos cuando el sistema está lleno de procesos bloqueados.
- **Bloqueado/Suspendido → Bloqueado.** La incursión de esta transición puede parecer propia de un diseño más bien pobre. Después de todo, si hay un proceso que no está listo para ejecutar y que no está en memoria principal, ¿qué sentido tiene traerlo? Pero considérese el siguiente escenario: un proceso termina, liberando alguna memoria principal. Hay un proceso en la cola de Bloqueados/Suspendidos con mayor prioridad que todos los procesos en la cola de Listos/Suspendidos y el sistema operativo tiene motivos para creer que el evento que lo bloquea va a ocurrir en breve. Bajo estas circunstancias, sería razonable traer el proceso Bloqueado a memoria por delante de los procesos Listos.
- **Ejecutando → Listo/Suspendido.** Normalmente, un proceso en ejecución se mueve a la cola de Listos cuando su tiempo de uso del procesador finaliza. Sin embargo, si el sistema operativo expulsa al proceso en ejecución porque un proceso de mayor prioridad en la cola de Bloqueado/Suspendido acaba de desbloquearse, el sistema operativo puede mover al proceso en ejecución directamente a la cola de Listos/Suspendidos y liberar parte de la memoria principal.
- **De cualquier estado → Saliente.** Habitualmente, un proceso termina cuando está ejecutando, bien porque ha completado su ejecución o debido a una condición de fallo. Sin embargo, en algunos sistemas operativos un proceso puede terminarse por el proceso que lo creó o cuando el proceso padre a su vez ha terminado. Si se permite esto, un proceso en cualquier estado puede moverse al estado Saliente.

Otros usos para la suspensión de procesos. Hace poco, hemos definido el concepto de procesos suspendidos como el proceso que no se encuentra en memoria principal. Un proceso que no está en memoria principal no está disponible de forma inmediata para su ejecución, independientemente de si está a la espera o no de un evento.

Podemos analizar el concepto de procesos suspendidos, definiendo un proceso suspendido como el que cumple las siguientes características:

1. El proceso no está inmediatamente disponible para su ejecución.

2. El proceso puede estar o no a la espera de un evento, si es así, la condición de bloqueo es independiente de la condición estar suspendido, y si sucede el evento que lo bloquea, eso no habilita al proceso para su ejecución inmediata.
3. El proceso fue puesto en estado suspendido por un agente: bien el proceso mismo, el proceso padre o el sistema operativo, con el propósito de prevenir su ejecución.
4. El proceso no puede ser recuperado de este estado hasta que el agente explícitamente así lo indique.

Tabla 3.3. Razones para la suspensión de un proceso.

Swapping	El sistema operativo necesita liberar suficiente memoria principal para traer un proceso en estado Listo de ejecución.
Otras razones del sistema operativo	El sistema operativo puede suspender un proceso en segundo plano o de utilidad o un proceso que se sospecha puede causar algún problema.
Solicitud interactiva del usuario	Un usuario puede desear suspender la ejecución de un programa con motivo de su depuración o porque está utilizando un recurso.
Temporización	Un proceso puede ejecutarse periódicamente (por ejemplo, un proceso monitor de estadísticas sobre el sistema) y puede suspenderse mientras espera el siguiente intervalo de ejecución.
Solicitud del proceso padre	Un proceso padre puede querer suspender la ejecución de un descendiente para examinar o modificar dicho proceso suspendido, o para coordinar la actividad de varios procesos descendientes.

La Tabla 3.3 muestra una lista de las condiciones para la suspensión de un proceso. Una de las razones que hemos discutido anteriormente es proporcionar espacio de memoria para traer procesos en estado Listos/Suspendidos o para incrementar el espacio de memoria disponible para los procesos Listos. El sistema operativo puede tener otros motivos para suspender un proceso. Por ejemplo, se puede utilizar un proceso de traza o de auditoría para monitorizar una actividad del sistema; el proceso puede recoger datos a nivel de utilización de varios recursos (procesador, memoria, canales) y la tasa de progreso del uso de los procesos en el sistema. El sistema operativo, bajo el control del operador, puede activar y desactivar este proceso con determinada periodicidad. Si el sistema operativo detecta o sospecha que puede haber un problema, puede suspender procesos. Un ejemplo de esto es un interbloqueo, que ya se discutirá en el Capítulo 6. Otro ejemplo, se detectan problemas en una línea de comunicación, el operador solicita al sistema operativo que suspenda el proceso que utiliza dicha línea hasta realizar algunas pruebas.

Otro tipo de razones están relacionadas con el uso interactivo del sistema. Por ejemplo, si los usuarios sospechan de un programa con algún tipo de *bug*, se puede detener la ejecución de dicho programa, examinando y modificando el programa y sus datos y reanudándolo de nuevo. O puede haber un proceso en segundo plano que recolecta trazas y estadísticas, el cual puede ser activado por parte del usuario.

Algunas consideraciones de tiempo también pueden llevar a activar decisiones de *swapping*. Por ejemplo, si un proceso se activa periódicamente pero está ocioso la mayor parte del tiempo puede ser enviado a disco entre cada una de las ejecuciones que realiza. Un ejemplo es un programa que monitoriza la utilización del sistema y la actividad del usuario.

Por último, un proceso padre puede suspender un proceso descendiente. Por ejemplo, el proceso A lanza al proceso B para realizar una lectura sobre un fichero. Posteriormente, el proceso B encuentra un error en la lectura del fichero y se lo indica al proceso A. El proceso A suspende al proceso B e investiga la causa.

En todos estos casos, la activación de un proceso Suspendido se solicita por medio del agente que inicialmente puso al proceso en suspensión.

3.3. DESCRIPCIÓN DE PROCESOS

El sistema operativo controla los eventos dentro del computador, planifica y activa los procesos para su ejecución por el procesador, reserva recursos para los mismos y responde a las solicitudes de servicios básicos de los procesos de usuario. Fundamentalmente, se piensa en el sistema operativo como en la entidad que gestiona el uso de recursos del sistema por parte de los procesos.

Este concepto se ilustra en la Figura 3.10. En un entorno multiprogramado, hay numerosos procesos (P1,... Pn) creados y residentes en memoria virtual. Cada proceso, durante el transcurso de su ejecución, necesita acceder a ciertos recursos del sistema, incluido el procesador, los dispositivos de E/S y la memoria principal. En la figura, el proceso P1 está ejecutando; al menos parte del proceso está en memoria principal, y controla dos dispositivos de E/S. El proceso P2 está también totalmente en memoria principal pero está bloqueado esperando a disponer de un dispositivo de E/S que está asignado a P1. El proceso Pn se encuentra transferido a disco (*swapping*) y está por tanto suspendido.

Exploraremos los detalles de la gestión de estos recursos por parte del sistema operativo en los próximos capítulos. Aquí nos interesa una cuestión más fundamental: ¿qué información necesita el sistema operativo para controlar los procesos y gestionar los recursos de estos?

ESTRUCTURAS DE CONTROL DEL SISTEMA OPERATIVO

Si el sistema operativo se encarga de la gestión de procesos y recursos, debe disponer de información sobre el estado actual de cada proceso y cada recurso. El mecanismo universal para proporcionar esta información es el siguiente: el sistema operativo construye y mantiene tablas de información sobre cada entidad que gestiona. Se indica una idea general del alcance de esta tarea en la Figura 3.11, que muestra cuatro diferentes tipos de tablas mantenidas por el sistema operativo: memoria, E/S, ficheros y procesos. A pesar de que los detalles difieren de un sistema operativo a otro, fundamentalmente, todos los sistemas operativos mantienen información de estas cuatro categorías.

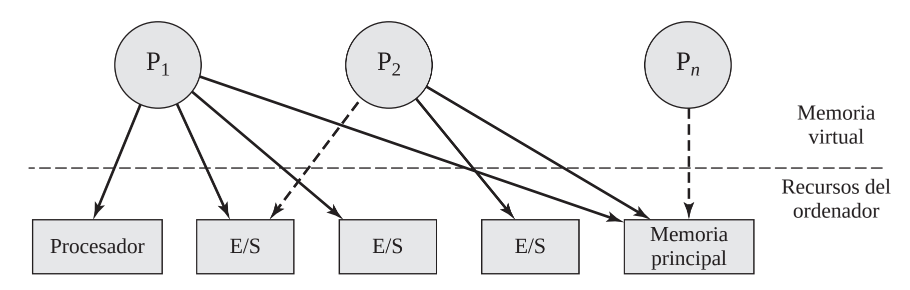


Figura 3.10. Procesos y recursos (reserva de recursos en una instantánea del sistema).

Las **tablas de memoria** se usan para mantener un registro tanto de la memoria principal (real) como de la secundaria (virtual). Parte de la memoria principal está reservada para el uso del sistema operativo; el resto está disponible para el uso de los procesos. Los procesos se mantienen en memoria secundaria utilizando algún tipo de memoria virtual o técnicas de *swapping*. Las tablas de memoria deben incluir la siguiente información:

- Las reservas de memoria principal por parte de los procesos.
- Las reservas de memoria secundaria por parte de los procesos.
- Todos los atributos de protección que restringe el uso de la memoria principal y virtual, de forma que los procesos puedan acceder a ciertas áreas de memoria compartida.
- La información necesaria para manejar la memoria virtual.

Veremos en detalle las estructuras de información para la gestión de memoria en la Parte Tres.

El sistema operativo debe utilizar las **tablas de E/S** para gestionar los dispositivos de E/S y los canales del computador. Pero, en un instante determinado, un dispositivo E/S puede estar disponible o asignado a un proceso en particular. Si la operación de E/S se está realizando, el sistema operativo necesita conocer el estado de la operación y la dirección de memoria principal del área usada como fuente o destino de la transferencia de E/S. La gestión de E/S se examina en detalle en el Capítulo 11.

El sistema operativo también puede mantener las **tablas de ficheros**. Estas tablas proporcionan información sobre la existencia de ficheros, su posición en almacenamiento secundario, su estado ac-

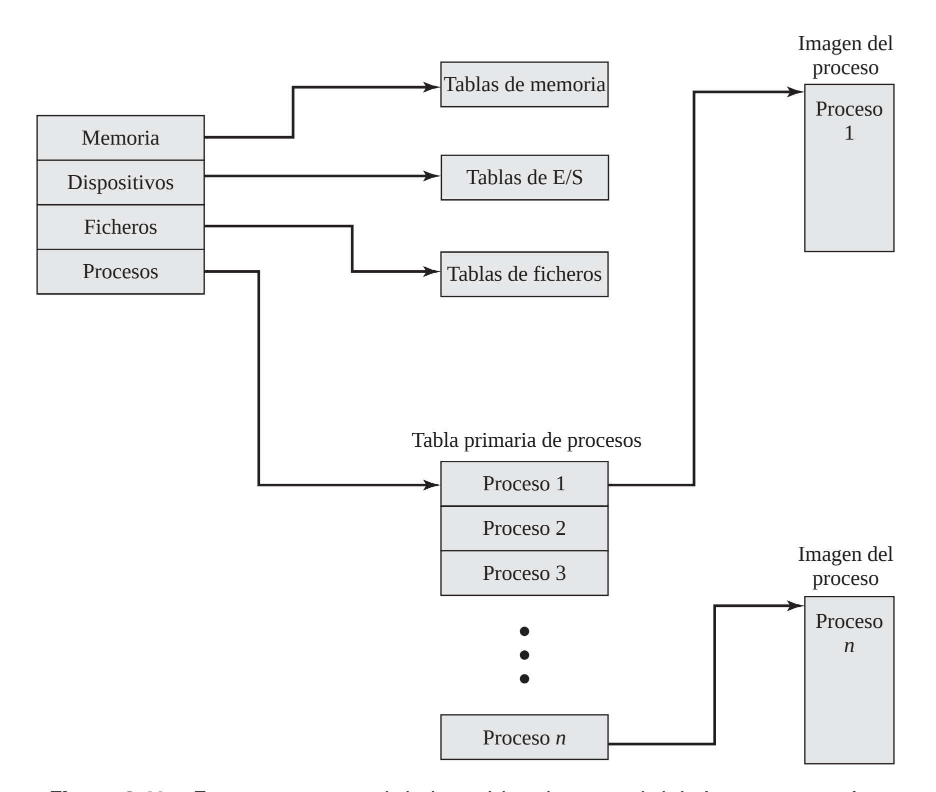


Figura 3.11. Estructura general de las tablas de control del sistema operativo.

tual, y otros atributos. La mayoría de, o prácticamente toda, esta información se puede gestionar por el sistema de ficheros, en cuyo caso el sistema operativo tiene poco o ningún conocimiento sobre los ficheros. En otros sistemas operativos, la gran mayoría del detalle de la gestión de ficheros sí que gestiona en el propio sistema operativo. Este aspecto se explorará en el Capítulo 12.

En último lugar, el sistema operativo debe mantener **tablas de procesos** para gestionar los procesos. La parte restante de esta sección se encuentra dedicada a examinar los requisitos que tienen las tablas de procesos. Antes de proceder con esta explicación, vamos a realizar dos puntualizaciones importantes. Primero, a pesar de que la Figura 3.11 muestra cuatro tipos diferentes de tablas debe quedar claro que estas tablas se encuentran entrelazadas y referenciadas entre sí de alguna manera. Memoria, E/S, y ficheros se gestionan por parte de los procesos, de forma que debe haber algunas referencias en estos recursos, directa o indirectamente, desde las tablas de procesos. Los ficheros indicados en las tablas de ficheros son accesibles mediante dispositivos E/S y estarán, en algún caso, residentes en memoria virtual o principal. Las tablas son también accesibles para el sistema operativo, y además están controladas por la gestión de memoria.

Segundo, ¿cómo puede el sistema operativo crear las tablas iniciales? Ciertamente, el sistema operativo debe tener alguna información sobre el entorno básico, tal como: cuánta memoria física existe, cuáles son los dispositivos de E/S y cuáles son sus identificadores, por ejemplo. Esto es una cuestión de configuración. Esto es, cuando el sistema operativo se inicializa, debe tener acceso a algunos datos de configuración que definen el entorno básico y estos datos se deben crear fuera del sistema operativo, con asistencia humana o por medio de algún software de autoconfiguración.

ESTRUCTURAS DE CONTROL DE PROCESO

Se considerará qué información debe conocer el sistema operativo si quiere manejar y controlar los procesos. Primero, debe conocer dónde están localizados los procesos, y segundo, debe conocer los atributos de los procesos que quiere gestionar (por ejemplo, identificador de proceso y estado del mismo).

Localización de los procesos. Antes de que podamos tratar la cuestión de dónde se encuentran los procesos o cuáles son sus atributos, debemos tratar una cuestión más fundamental: ¿qué es la representación física de un proceso? Como mínimo, un proceso debe incluir un programa o un conjunto de programas a ejecutar. Asociados con estos programas existen unas posiciones de memoria para los datos de variables locales y globales y de cualquier constante definida. Así, un proceso debe consistir en una cantidad suficiente de memoria para almacenar el programa y datos del mismo. Adicionalmente, la ejecución típica de un programa incluye una pila (*véase* Apéndice 1B) que se utiliza para registrar las llamadas a procedimientos y los parámetros pasados entre dichos procedimientos. Por último, cada proceso está asociado a un número de atributos que son utilizados por el sistema operativo para controlar el proceso. Normalmente, el conjunto de estos atributos se denomina **bloque de control del proceso** (BCP)[6]. Nos podemos referir al conjunto de programa, datos, pila, y atributos, como la **imagen del proceso** (Tabla 3.4).

La posición de la imagen del proceso dependerá del esquema de gestión de memoria que se utilice. En el caso más simple, la imagen del proceso se mantiene como un bloque de memoria contiguo, o continuo. Este bloque se mantiene en memoria secundaria, habitualmente disco. Para que el sistema operativo pueda gestionar el proceso, al menos una pequeña porción de su imagen se debe mantener

[6] Otros nombres habituales usados para esta estructura de datos son bloque de control de tarea, descriptor del proceso, y descriptor de tarea.

en memoria principal. Para ejecutar el proceso, la imagen del proceso completa se debe cargar en memoria principal o al menos en memoria virtual. Asimismo, el sistema operativo necesita conocer la posición en disco de cada proceso y, para cada proceso que se encuentre en memoria principal, su posición en dicha memoria. Se vio una ligera modificación un poco más compleja de este esquema con el sistema operativo CTSS, en el Capítulo 2. Con CTSS, cuando un proceso se transfiere a disco (*swapping*) parte de la imagen del proceso permanece en memoria principal. De esta forma, el sistema operativo debe mantener en registro qué partes de la imagen del proceso se encuentran todavía en memoria principal.

Tabla 3.4. Elementos típicos de la imagen de un proceso.

Datos del usuario	La parte modificable del espacio de usuario. Puede incluir datos de programa, área de pila de usuario, y programas que puedan ser modificados.
Programa de usuario	El programa a ejecutar
Pila de sistema	Cada proceso tiene una o más pilas de sistema (LIFO) asociadas a él. Una pila se utiliza para almacenar los parámetros y las direcciones de retorno de los procedimientos y llamadas a sistema.
Bloque de control de proceso	Datos necesarios para que el sistema operativo pueda controlar los procesos (*véase* tabla 3.5).

Los sistemas operativos modernos suponen la existencia de un hardware de paginación que permite el uso de la memoria física no contigua, para dar soporte a procesos parcialmente residentes en la memoria principal. En esos casos, una parte de la imagen del proceso se puede encontrar en dicha memoria principal, con las restantes en memoria secundaria[7]. De esta forma, las tablas mantenidas por sistema operativo deben mostrar la localización de cada página de la imagen del proceso.

La Figura 3.11 muestra la estructura de la información de localización de la siguiente forma. Hay una tabla primaria de procesos con una entrada por cada proceso. Cada entrada contiene, al menos, un puntero a la imagen del proceso. Si la imagen del proceso contiene múltiples bloques, esta información se mantiene directamente en la tabla principal del proceso o bien está disponible por referencias cruzadas entre las entradas de las tablas de memoria. Por supuesto, esta es una representación genérica; un sistema operativo en particular puede tener su propia forma de organizar esta información de localización.

Atributos de proceso. Cualquier sistema operativo multiprogramado de la actualidad requiere una gran cantidad de información para manejar cada proceso. Como quedó explicado, esta información se puede considerar que reside en el bloque de control del proceso (BCP). Los diferentes sistemas organizarán esta información de formas diferentes. Al final de este capítulo y en el próximo se muestran varios ejemplos. Por ahora, exploraremos simplemente el tipo de información que el sistema operativo puede utilizar, sin considerar cualquier detalle de cómo esta información está organizada.

[7] Esta breve explicación pasa por encima algunos detalles. En particular, en un sistema que utilice memoria virtual, toda imagen de un proceso activo se encuentra siempre en memoria secundaria. Sólo una parte de la imagen se carga en memoria principal, ésta se copia en lugar de moverse. De esta forma, la memoria secundaria mantiene una copia de todos los segmentos y/o páginas. Sin embargo, si la parte de imagen del proceso en memoria principal se ve modificada, la copia en memoria secundaria estará desactualizada hasta que la parte residente en memoria principal se copie de nuevo al disco.

La Tabla 3.5 muestra la lista de los tipos de categorías de información que el sistema operativo requiere para cada proceso. Resulta sorprendente la cantidad de información necesaria para esta gestión. Cuando se vaya apreciando con mayor exactitud las responsabilidades del sistema operativo, esta lista parecerá más razonable.

Podemos agrupar la información del bloque de control del proceso en tres categorías generales:

- Identificación del proceso
- Información de estado del procesador
- Información de control del proceso

Tabla 3.5. Elementos típicos de un bloque de control del proceso (BCP).

Identificación del proceso

Identificadores

Identificadores numéricos que se pueden guardar dentro del bloque de control del proceso:

- identificadores del proceso
- identificador del proceso que creó a este proceso (proceso padre)
- identificador del usuario

Información de estado del procesador

Registros visibles por el usuario

Un registro visible por el usuario es aquel al que se puede hacer referencia por medio del lenguaje máquina que ejecuta el procesador cuando está en modo usuario. Normalmente, existen de 8 a 32 de estos registros, aunque determinadas implementaciones RISC tienen más de 100.

Registros de estado y control

Hay una gran variedad de registros del procesador que se utilizan para el control de operaciones. Estos incluyen:

- *Contador de programa*: contiene la dirección de la siguiente instrucción a ejecutar.
- *Códigos de condición*: resultan de la operación lógica o aritmética más reciente (por ejemplo, signo, cero, acarreo, igual, desbordamiento).
- *Información de estado*: incluyen los *flags* de interrupciones habilitadas/deshabilitadas, modo ejecución.

Puntero de pila

Cada proceso tiene una o más pilas de sistema (LIFO) asociadas a él. Una pila se utiliza para almacenar los parámetros y las direcciones de retorno de los procedimientos y llamadas a sistema. Un puntero de pila apunta a la parte más alta de la pila.

Información de control de proceso

Información de estado y de planificación

Esta es la información que el sistema operativo necesita para analizar las funciones de planificación. Elementos típicos de esta información son:

- *Estado del proceso*: indica si está listo o no el proceso para ser planificado para su ejecución (por ejemplo, Ejecutando, Listo, Esperando, Detenido).
- *Prioridad*: uno o más campos que se pueden utilizar para escribir la prioridad de planificación del proceso. En algunos sistemas, se necesitan múltiples valores (por ejemplo, por-defecto, actual, mayor-disponible).
- *Información relativa a planificación*: esto dependerá del algoritmo de planificación utilizado. Por ejemplo, la cantidad de tiempo que el proceso estaba esperando y la cantidad de tiempo que el proceso ha ejecutado la última vez que estuvo corriendo.
- *Evento*: identificar el evento al cual el proceso está esperando para continuar su ejecución.

Estructuración de datos

Un proceso puede estar enlazado con otro proceso en una cola, anillo o con cualquier otra estructura. Por ejemplo, todos los procesos en estado de espera por un nivel de prioridad en particular pueden estar enlazados en una cola. Un proceso puede mostrar una relación padre-hijo (creador-creado) con otro proceso. El bloque de control del proceso puede contener punteros a otros procesos para dar soporte a estas estructuras.

Comunicación entre procesos

Se pueden asociar diferentes *flags*, señales, y mensajes relativos a la comunicación entre dos procesos independientes. Alguna o toda esta información se puede mantener en el bloque de control de proceso (BCP).

Privilegios de proceso

Los procesos adquieren privilegios de acuerdo con la memoria a la que van a acceder y los tipos de instrucciones que se pueden ejecutar. Adicionalmente, los privilegios se pueden utilizar para usar utilidades de sistema o servicios.

Gestión de memoria

Esta sección puede incluir punteros a tablas de segmentos y/o páginas que describen la memoria virtual asignada a este proceso.

Propia de recursos y utilización

Se deben indicar los recursos controlados por el proceso, por ejemplo, ficheros abiertos. También se puede incluir un histórico de utilización del procesador o de otros recursos; esta información puede ser necesaria para el planificador.

Con respecto al **identificador de proceso**, en prácticamente todos los sistemas operativos, a cada proceso se le asocia un identificador numérico único, que puede ser simplemente un índice en la tabla de procesos principal (Figura 3.11); de otra forma, debe existir una traducción que permita al sistema operativo localizar las tablas apropiadas basándose en dicho identificador de proceso. Este identificador es útil para diferentes cuestiones. Muchas de las otras tablas controladas por el sistema operativo incluyen información que referencia a otro proceso por medio del identificador de proceso. Por ejemplo, se puedan organizar las tablas de memoria para proporcionar un mapa de la memoria principal que indique a qué proceso se tiene asignada cada región. Pueden aparecer referencias similares en las tablas de E/S o de ficheros. Cuando un proceso se comunica con otro proceso, el identificador de proceso informa al sistema operativo del destino de la comunicación. Cuando los procesos pueden crear otros procesos, los identificadores indican el proceso padre y los descendientes en cada caso.

Junto con estos identificadores de proceso, a un proceso se le puede asignar un identificador de usuario que indica qué usuario es responsable del trabajo.

La **información de estado de proceso** indica los contenidos de los registros del procesador. Cuando un proceso está ejecutando, esta información está, por supuesto, en los registros. Cuando un proceso se interrumpe, toda la información de los registros debe salvaguardarse de forma que se pueda restaurar cuando el proceso continúe con su ejecución. La naturaleza y el número de estos registros depende del diseño del procesador. Normalmente, el conjunto de registros incluye registros visibles por usuario, registros de control y de estado, y punteros de pila. Todos estos se describieron en el Capítulo 1.

Se debe notar que, el diseño de todos los procesadores incluye un registro o conjuntos de registros, habitualmente conocidos como palabras de estado de programa (*program status word*, PSW) que contiene información de estado. La PSW típicamente contiene códigos de condición así como otra información de estado. Como ejemplo de una palabra de estado de un procesador se tiene lo que los procesadores Pentium definen como registros EFLAFS (*véase* Figura 3.12 y Tabla 3.6). Esta estructura es utilizada por los sistemas operativos (incluyendo UNIX y Windows) que se ejecuten sobre procesadores de tipo Pentium.

La tercera categoría principal de información del bloque de control de proceso se denomina, a falta de otro nombre mejor, **información de control de proceso**. Esta información adicional la necesita el sistema operativo para controlar y coordinar varios procesos activos. La última parte de la Tabla 3.5 indica el ámbito de esta información. Según examinemos los detalles de funcionamiento del sistema operativo en los sucesivos capítulos, la necesidad de algunos de los elementos de esta lista parecerá más clara.

El contenido de esta figura no debería modificarse. La traducción de los registros de control de una arquitectura en concreto no tiene mucho sentido, puesto que las siglas con las que se denotan están en inglés. El significado de dichas siglas y la traducción del término se encuentran en la siguiente tabla.

La Figura 3.13 sugiere la estructura de las imágenes de los procesos en memoria virtual. Cada imagen de proceso consiste en un bloque de control de proceso, una pila de usuario, un espacio de direcciones privadas del proceso, y cualquier otro espacio de direcciones que el proceso comparta con otros procesos. En la figura, cada imagen de proceso aparece como un rango de direcciones contiguo, pero en una implementación real, este no tiene por qué ser el caso; dependerá del esquema de gestión de memoria y de la forma en la cual se encuentren organizadas las estructuras de control por parte del sistema operativo.

31	21					16	15														0
	ID	VIP	VIF	AC	VM	RF		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF

ID = (Indicador de identificación)
VIP = (Interrupción virtual pendiente)
VIF = (Indicador de interrupción virtual)
AC = (Comprobación de alineamiento)
VM = (Modo virtual 8086)
RF = (Indicación de reanudación)
NT = (Indicador de tarea anidada)
IOPL = (Nivel de privilegio de E/S)
OF = (Indicador de desbordamiento)
DF = (Indicador de dirección)
IF = (Indicador de interrupciones habilitadas)
TF = (Indicador de *trap*)
SF = (Indicador de signo)
ZF = (Indicador de cero)
AF = Indicador auxiliar de acarreo)
PF = (Indicador de paridad)
CF = (Indicador de acarreo)

Figura 3.12. Registro EFLAGS de un Pentium II.

Tabla 3.6. Bits del registro EFLAGS de un Pentium.

Bits de control	
AC (*Alignment check*)	Fija si una palabra (o una doble-palabra) se encuentra direccionada en la frontera entre palabras (o dobles-palabras).
ID (*Identification flag*)	Si este bit puede ser puesto a 1 y a 0, este procesador soporta la instrucción CPUID. Esta instrucción proporciona información sobre el vendedor, familia, y modelo.
RF (*Resume flag*)	Permite al programador desactivar las extensiones de depuración de forma que la instrucción pueda volverse a ejecutar después de una excepción de depuración sin causar inmediatamente después otra excepción de depuración.
IOPL (*I/O privilege level*)	Cuando está activado, hace que el procesador genere una excepción para todos los accesos a dispositivo de E/S durante una operación en modo protegido.
DF (*Direction flag*)	Determina cuando una instrucción de procesamiento de cadenas de caracteres incrementa o decrementa los semi-registros de 16 bits SE y DI (para operaciones de 16 bits) o los registros de 32 bits ESE y EDI (para operaciones de 32 bits).
IF (*Interruption enable flag*)	Cuando está puesto a 1, el procesador reconocerá interrupciones externas.
TF (*Trap flag*)	Cuando está puesto a 1, causa una interrupción después de la ejecución de cada instrucción. Su uso está dirigido a la depuración de código.
Bits de modos de operación	
NT (*Nested task flag*)	Indica que la tarea actual está anidada dentro de otra tarea en modo de operación protegido.
VM (*Virtual 8086 mode*)	Permite al programador activar o desactivar el modo virtual 8086, que determina si el procesador funciona como si fuese una máquina 8086.
VIP (*Virtual interrupt pending*)	Usado en el modo virtual 8086 para indicar que una o más interrupciones están a la espera de servicio.
VIF (*Virtual interruption flag*)	Usado en modo virtual 8086 en lugar de IF.
Códigos de condición	
AF (*Auxiliar carry flag*)	Representa el acarreo o resto entre medios bytes de una operación aritmética o lógica de 8 bits usando el registro AL.
CF (*Carry flag*)	Indica el acarreo o el resto por medio del bit situado más a la izquierda después de una operación aritmética. También se modificaron por algunas operaciones de desplazamiento o rotación.
OF (*Overflow flag*)	Indica un desbordamiento (*overflow*) aritmético después de una suma o resta.
PF (*Parity flag*)	Paridad del resultado de una operación aritmética o lógica. 1 indica paridad par; 0 indica paridad impar.
SF (*Sign flag*)	Indica el signo del resultado de una operación aritmética o lógica.
ZF (*Zero flag*)	Indica que el resultado de una operación aritmética o lógica es 0.

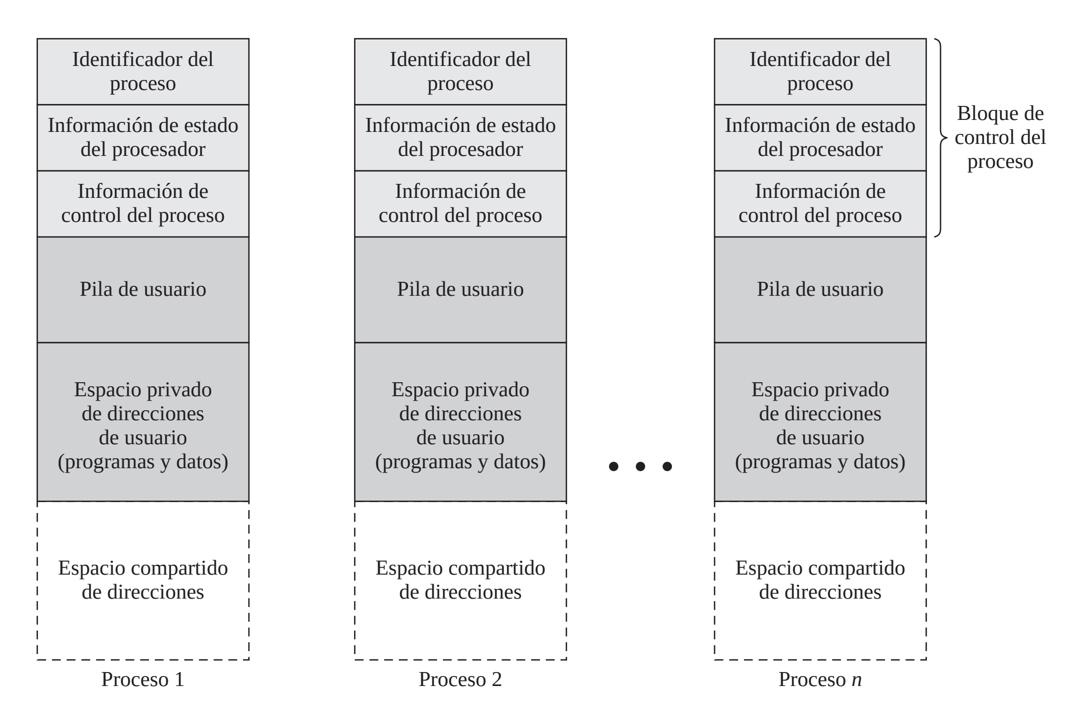


Figura 3.13. Procesos de usuario en memoria virtual.

Como indica la Tabla 3.5, un bloque de control de proceso puede contener información estructural, incluyendo punteros que permiten enlazar bloques de control de proceso entre sí. De esta forma, las colas que se describieron en la sección anterior pueden implementarse como listas enlazadas de bloques de control de proceso. Por ejemplo, la estructura de colas de la Figura 3.8a se puede implementar tal y como sugiere la Figura 3.14.

El papel del bloque de control de proceso. El bloque de control de proceso es la más importante de las estructuras de datos del sistema operativo. Cada bloque de control de proceso contiene toda la información sobre un proceso que necesita el sistema operativo. Los bloques se leen y/o se modifican por la práctica totalidad de los módulos del sistema operativo, incluyendo aquellos relacionados con la planificación, la reserva de recursos, el procesamiento entre regiones, y el análisis y monitorización del rendimiento. Se puede decir que el conjunto de los bloques de control de proceso definen el estado del sistema operativo.

Esto muestra un aspecto importante en el diseño. Un gran número de rutinas dentro del sistema operativo necesitarán acceder a información de los bloques de control de proceso. Proporcionar acceso directo a estas tablas no es difícil. Cada proceso lleva asociado un único identificador, que puede utilizarse para indexar dentro de una tabla de punteros a bloques de control de proceso. La dificultad no reside en el acceso sino en la protección. Dos posibles problemas se pueden presentar son:

- Un fallo en una simple rutina, como un manejador de interrupción, puede dañar los bloques de control de proceso, que puede destruir la capacidad del sistema para manejar los procesos afectados.
- Un cambio en la estructura o en la semántica de los bloques de control de procesos puede afectar a un gran número de módulos del sistema operativo.

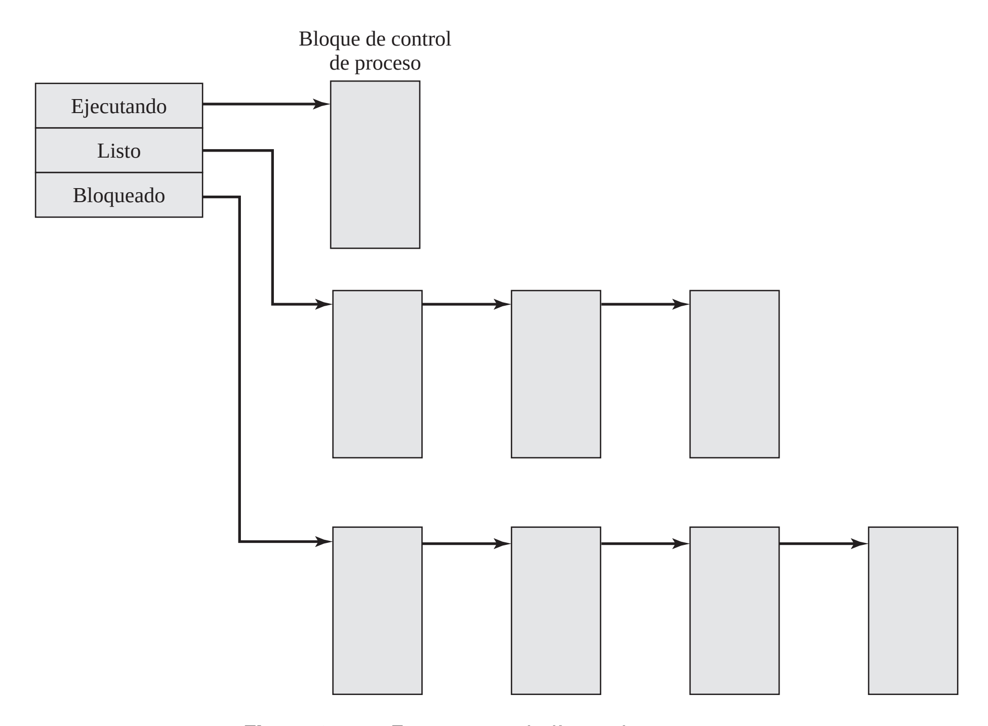


Figura 3.14. Estructuras de listas de procesos.

Estos problemas se pueden ser tratar obligando a que todas rutinas del sistema operativo pasen a través de una rutina de manejador, cuyo único trabajo es proteger a los bloques de control de proceso, y que es la única que realiza el arbitraje en las operaciones de lectura y escritura de dichos bloques. Los factores a equilibrar en el uso de está rutina son, por un lado los aspectos de rendimiento, y por el otro, el grado en el cual el resto del software del sistema puede verificarse para determinar su corrección.

3.4. CONTROL DE PROCESOS

MODOS DE EJECUCIÓN

Antes de continuar con nuestra explicación sobre cómo el sistema operativo gestiona los procesos, necesitamos distinguir entre los modos de ejecución del procesador, normalmente asociados con el sistema operativo y los asociados con los programas de usuario. Muchos procesadores proporcionan al menos dos modos de ejecución. Ciertas instrucciones se pueden ejecutar en modos privilegiados únicamente. Éstas incluirían lectura y modificación de los registros de control, por ejemplo la palabra de estado de programa; instrucciones de E/S primitivas; e instrucciones relacionadas con la gestión de memoria. Adicionalmente, ciertas regiones de memoria sólo se pueden acceder en los modos más privilegiados.

El modo menos privilegiado a menudo se denomina **modo usuario**, porque los programas de usuario típicamente se ejecutan en este modo. El modo más privilegiado se denomina **modo sistema**, **modo control** o **modo núcleo**. Este último término se refiere al núcleo del sistema operativo, que es la parte del sistema operativo que engloba las funciones más importantes del sistema. La Tabla 3.7 lista las funciones que normalmente se encuentran dentro del núcleo del sistema operativo.

Tabla 3.7. Funciones típicas de un núcleo de sistema operativo.

Gestión de procesos

- Creación y terminación de procesos
- Planificación y activación de procesos
- Intercambio de procesos
- Sincronización de procesos y soporte para comunicación entre procesos
- Gestión de los bloques de control de proceso

Gestión memoria

- Reserva de espacio direcciones para los procesos
- *Swapping*
- Gestión de páginas y segmentos

Gestión E/S

- Gestión de *buffers*
- Reserva de canales de E/S y de dispositivos para los procesos

Funciones de soporte

- Gestión de interrupciones
- Auditoría
- Monitorización

El motivo por el cual se usan los otros modos es claro. Se necesita proteger al sistema operativo y a las tablas clave del sistema, por ejemplo los bloques de control de proceso, de la interferencia con programas de usuario. En modo núcleo, el software tiene control completo del procesador y de sus instrucciones, registros, y memoria. Este nivel de control no es necesario y por seguridad no es recomendable para los programas de usuario.

Aparecen dos cuestiones: ¿cómo conoce el procesador en que modo está ejecutando y cómo este modo puede modificarse? En lo referente la primera cuestión, existe típicamente un bit en la palabra de estado de programa (PSW) que indica el modo de ejecución. Este bit se cambia como respuesta a determinados eventos. Habitualmente, cuando un usuario realiza una llamada a un servicio del sistema operativo o cuando una interrupción dispara la ejecución de una rutina del sistema operativo, este modo de ejecución se cambia a modo núcleo y; tras la finalización del servicio, el modo se fija de nuevo a modo usuario. Por ejemplo, si consideramos el procesador Intel Itanium, que implementa la arquitectura IA-64 de 64 bits, el procesador tiene un registro de estado (psr) que incluye un campo de los 2 bits llamado cpl (*current privilege level*). El Nivel 0 es en el nivel con más privilegios mientras que el Nivel 3 es el nivel con menos privilegios. La mayoría de los sistemas operativos, como Linux, utilizar el Nivel 0 para el núcleo y uno de los otros niveles como nivel de usuario. Cuando llega una interrupción, el procesador limpia la mayor parte de los bits en psr, incluyendo el campo cpl. Esto automáticamente pone en 0 el cpl. Al final de la rutina de manejo de interrupción, la última instrucción que se ejecuta es irt (interrupt return). Esta instrucción hace que el procesador restaure el psr del programa interrumpido, que restaura a su vez el nivel de privilegios de dicho programa. Una secuencia similar ocurre cuando la aplicación solicita una llamada al sistema. Para los Itanium, una aplicación solicita la llamada sistema indicando un identificador de llamada de sistema

y los argumentos de dicha llamada en unas áreas definidas y posteriormente ejecutando una instrucción especial que se puede invocar en modo usuario y que tiene como objeto transferir el control al núcleo.

CREACIÓN DE PROCESOS

En la Sección 3.1 hemos comentado los eventos que llevará creación de uno proceso. Una vez vistas las estructuras de datos asociadas a un proceso, ahora posiciones está en posición de describir brevemente los pasos que llevan a la verdadera creación de un proceso.

Una vez que el sistema operativo decide, por cualquier motivo (Tabla 3.1), crear un proceso procederá de la siguiente manera:

1. **Asignar un identificador de proceso único al proceso**. En este instante, se añade una nueva entrada a la tabla primaria de procesos, que contiene una entrada por proceso.
2. **Reservar espacio para proceso**. Esto incluye todos los elementos de la imagen del proceso. Para ello, el sistema operativo debe conocer cuánta memoria se requiere para el espacio de direcciones privado (programas y datos) y para la pila de usuario. Estos valores se pueden asignar por defecto basándonos en el tipo de proceso, o pueden fijarse en base a la solicitud de creación del trabajo remitido por el usuario. Si un proceso es creado por otro proceso, el proceso padre puede pasar los parámetros requeridos por el sistema operativo como parte de la solicitud de la creación de proceso. Si existe una parte del espacio direcciones compartido por este nuevo proceso, se fijan los enlaces apropiados. Por último, se debe reservar el espacio para el bloque de control de proceso (BCP).
3. **Inicialización del bloque de control de proceso**. La parte de identificación de proceso del BCP contiene el identificador del proceso así como otros posibles identificadores, tal como el indicador del proceso padre. En la información de estado de proceso del BCP, habitualmente se inicializa con la mayoría de entradas a 0, excepto el contador de programa (fijado en el punto entrada del programa) y los punteros de pila de sistema (fijados para definir los límites de la pila del proceso). La parte de información de control de procesos se inicializa en base a los valores por omisión, considerando también los atributos que han sido solicitados para este proceso. Por ejemplo, el estado del proceso se puede inicializar normalmente a Listo o Listo/Suspendido. La prioridad se puede fijar, por defecto, a la prioridad más baja, a menos que una solicitud explicita la eleve a una prioridad mayor. Inicialmente, el proceso no debe poseer ningún recurso (dispositivos de E/S, ficheros) a menos que exista una indicación explícita de ello o que haya sido heredados del padre.
4. **Establecer los enlaces apropiados**. Por ejemplo, si el sistema operativo mantiene cada cola del planificador como una lista enlazada, el nuevo proceso debe situarse en la cola de Listos o en la cola de Listos/Suspendidos.
5. **Creación o expansión de otras estructuras de datos**. Por ejemplo, el sistema operativo puede mantener un registro de auditoría por cada proceso que se puede utilizar posteriormente a efectos de facturación y/o de análisis de rendimiento del sistema.

CAMBIO DE PROCESO

A primera vista, la operación de cambio de proceso puede parecer sencilla. En algunos casos, un proceso en ejecución se interrumpe para que el sistema operativo asigne a otro proceso el estado de Ejecutando y de esta forma establecer el turno entre los procesos. Sin embargo, es necesario tener en

cuenta algunas consideraciones de diseño. Primero, ¿qué evento dispara los cambios de proceso? Otra consideración que se debe tener en cuenta es la distinción entre cambio de proceso (*process switching*) y cambio de modo (*mode switching*). Por último, ¿qué debe hacer el sistema operativo con las diferentes estructuras que gestiona para proporcionar un cambio de proceso?

Cuándo se realiza el cambio de proceso. Un cambio de proceso puede ocurrir en cualquier instante en el que el sistema operativo obtiene el control sobre el proceso actualmente en ejecución. La Tabla 3.8 indica los posibles momentos en los que el sistema operativo puede tomar dicho control.

Primero, consideremos las interrupciones del sistema. Realmente, podemos distinguir, como hacen muchos sistemas, dos tipos diferentes de interrupciones de sistema, unas simplemente denominadas interrupciones, y las otras denominadas *traps*. Las primeras se producen por causa de algún tipo de evento que es externo e independiente al proceso actualmente en ejecución, por ejemplo la finalización de la operación de E/S. Las otras están asociadas a una condición de error o excepción generada dentro del proceso que está ejecutando, como un intento de acceso no permitido a un fichero. Dentro de una **interrupción** ordinaria, el control se transfiere inicialmente al manejador de interrupción, que realiza determinadas tareas internas y que posteriormente salta a una rutina del sistema operativo, encargada de cada uno de los tipos de interrupciones en particular. Algunos ejemplos son:

- **Interrupción de reloj.** El sistema operativo determinar si el proceso en ejecución ha excedido o no la unidad máxima de tiempo de ejecución, denominada **rodaja de tiempo** (*time slice*). Esto es, una rodaja de tiempo es la máxima cantidad de tiempo que un proceso puede ejecutar antes de ser interrumpido. En dicho caso, este proceso se puede pasar al estado de Listo y se puede activar otro proceso.
- **Interrupción de E/S.** El sistema operativo determina qué acción de E/S ha ocurrido. Si la acción de E/S constituye un evento por el cual están esperando uno o más procesos, el sistema operativo mueve todos los procesos correspondientes al estado de Listos (y los procesos en estado Bloqueado/Suspendido al estado Listo/Suspendido). El sistema operativo puede decidir si reanuda la ejecución del proceso actualmente en estado Ejecutando o si lo expulsa para proceder con la ejecución de un proceso Listo de mayor prioridad.

Tabla 3.8. Mecanismos para la interrupción de la ejecución de un proceso.

Mecanismo	Causa	Uso
Interrupción	Externa a la ejecución del proceso actualmente en ejecución.	Reacción ante un evento externo asíncrono.
Trap	Asociada a la ejecución de la instrucción actual.	Manejo de una condición de error o de excepción.
Llamada al sistema	Solicitud explícita.	Llamada a una función del sistema operativo.

- **Fallo de memoria.** El procesador se encuentra con una referencia a una dirección de memoria virtual, a una palabra que no se encuentra en memoria principal. El sistema operativo debe traer el bloque (página o segmento) que contiene la referencia desde memoria secundaria a memoria principal. Después de que se solicita la operación de E/S para traer el bloque a memoria, el proceso que causó el fallo de memoria se pasa al estado de Bloqueado; el sistema operativo realiza un cambio de proceso y pone a ejecutar a otro proceso. Después de que el bloque de memoria solicitado se haya traído, el proceso pasará al estado Listo.

Con un ***trap***, el sistema operativo conoce si una condición de error o de excepción es irreversible. Si es así, el proceso en ejecución se pone en el estado Saliente y se hace un cambio de proceso. De no ser así, el sistema operativo actuará dependiendo de la naturaleza del error y su propio diseño. Se puede intentar un procedimiento de recuperación o simplemente la notificación al usuario, pudiendo implicar tanto un cambio de proceso como la continuación de la ejecución del proceso actual.

Por último, el sistema operativo se puede activar por medio de una **llamada al sistema** procedente del programa en ejecución. Por ejemplo, si se está ejecutando un proceso y se ejecuta una operación que implica una llamada de E/S, como abrir un archivo. Esta llamada implica un salto a una rutina que es parte del código del sistema operativo. La realización de una llamada al sistema puede implicar en algunos casos que el proceso que la realiza pase al estado de Bloqueado.

Cambio de modo. En el Capítulo 1, discutimos la inclusión de una fase de interrupción como parte del ciclo de una instrucción. Recordando esto, en la fase de interrupción, el procesador comprueba que no exista ninguna interrupción pendiente, indicada por la presencia de una señal de interrupción. Si no hay interrupciones pendientes, el procesador pasa a la fase de búsqueda de instrucción, siguiendo con el programa del proceso actual. Si hay una interrupción pendiente, el proceso actúa de la siguiente manera:

1. Coloca el contador de programa en la dirección de comienzo de la rutina del programa manejador de la interrupción.
2. Cambia de modo usuario a modo núcleo de forma que el código de tratamiento de la interrupción pueda incluir instrucciones privilegiadas.

El procesador, acto seguido, pasa a la fase de búsqueda de instrucción y busca la primera instrucción del programa de manejo de interrupción, que dará servicio a la misma. En este punto, habitualmente, el contexto del proceso que se ha interrumpido se salvaguarda en el bloque de control de proceso del programa interrumpido.

Una pregunta que se puede plantear es, ¿qué constituye el contexto que se debe salvaguardar? La respuesta es que se trata de toda la información que se puede ver alterada por la ejecución de la rutina de interrupción y que se necesitará para la continuación del proceso que ha sido interrumpido. De esta forma, se debe guardar la parte del bloque de control del proceso que hace referencia a la información de estado del procesador. Esto incluye el contador de programa, otros registros del procesador, y la información de la pila.

¿Se necesita hacer algo más? Eso depende de qué ocurra luego. El manejador de interrupción es habitualmente un pequeño programa que realiza unas pocas tareas básicas relativas a la interrupción. Por ejemplo, borra el *flag* o indicador que señala la presencia de interrupciones. Puede enviar una confirmación a la entidad que lanzó dicha interrupción, como por ejemplo el módulo de E/S. Y puede realizar algunas tareas internas variadas relativas a los efectos del evento que causó la interrupción. Por ejemplo, si la interrupción se refiere a un evento de E/S, el manejador de interrupción comprobará la existencia o no de una condición de error. Si ha ocurrido un error, el manejador mandará una señal al proceso que solicitó dicha operación de E/S. Si la interrupción proviene del reloj, el manejador la va a pasar el control al activador, el cual decidirá pasar a otro proceso debido a que la rodaja de tiempo asignada a ese proceso ha expirado.

¿Qué pasa con el resto de información del bloque de control de proceso? Si a esta interrupción le sigue un cambio de proceso a otro proceso, se necesita hacer algunas cosas más. Sin embargo, en muchos sistemas operativos, la existencia de una interrupción no implica necesariamente un cambio de proceso. Es posible, por tanto, que después de la ejecución de la rutina de interrupción, la ejecución se reanude con el mismo proceso. En esos casos sólo se necesita salvaguardar la información del estado del procesador cuando se produce la interrupción y restablecerlo cuando se reanude la

ejecución del proceso. Habitualmente, las operaciones de salvaguarda y recuperación se realizan por hardware.

Cambio del estado del proceso. Esta claro, por tanto, que el cambio de modo es un concepto diferente del cambio de proceso[8]. Un cambio de modo puede ocurrir sin que se cambie el estado del proceso actualmente en estado Ejecutando. En dicho caso, la salvaguarda del estado y su posterior restauración comportan sólo una ligera sobrecarga. Sin embargo, si el proceso actualmente en estado Ejecutando, se va a mover a cualquier otro estado (Listo, Bloqueado, etc.), entonces el sistema operativo debe realizar cambios sustanciales en su entorno. Los pasos que se realizan para un cambio de proceso completo son:

1. Salvar el estado del procesador, incluyendo el contador de programa y otros registros.
2. Actualizar el bloque de control del proceso que está actualmente en el estado Ejecutando. Esto incluye cambiar el estado del proceso a uno de los otros estados (Listo, Bloqueado, Listo/Suspendido, o Saliente). También se tienen que actualizar otros campos importantes, incluyendo la razón por la cual el proceso ha dejado el estado de Ejecutando y demás información de auditoría.
3. Mover el bloque de control de proceso a la cola apropiada (Listo, Bloqueado en el evento *i*, Listo/Suspendido).
4. Selección de un nuevo proceso a ejecutar; esta cuestión se analiza con más detalle en la Parte Cuatro.
5. Actualizar el bloque de control del proceso elegido. Esto incluye pasarlo al estado Ejecutando.
6. Actualizar las estructuras de datos de gestión de memoria. Esto se puede necesitar, dependiendo de cómo se haga la traducción de direcciones; estos aspectos se cubrirán en la Parte Tres.
7. Restaurar el estado del procesador al que tenía en el momento en el que el proceso seleccionado salió del estado Ejecutando por última vez, leyendo los valores anteriores de contador de programa y registros.

Por tanto, el cambio de proceso, que implica un cambio en el estado, requiere un mayor esfuerzo que un cambio de modo.

EJECUCIÓN DEL SISTEMA OPERATIVO

En el Capítulo 2, señalamos dos aspectos intrínsecos del sistema operativo:

- El sistema operativo funciona de la misma forma que cualquier software, en el sentido que un sistema operativo es un conjunto de programas ejecutados por un procesador.
- El sistema operativo, con frecuencia, cede el control y depende del procesador para recuperar dicho control.

---

[8] El término **cambio de contexto** (*context switch*) se usa en literatura sobre sistemas operativos y en libros de texto. Desafortunadamente, aunque la mayoría de estas referencias utilizan este término para lo que en este libro se ha denominado cambio de proceso, otras fuentes lo usan para referirse al cambio de modo o incluso al cambio de thread (definido en el siguiente capítulo). Para evitar ambigüedades, este término no se ha utilizado en este libro.

Si el sistema operativo es simplemente una colección de programas y si son ejecutados por el procesador, tal y como cualquier otro programa, ¿es el sistema operativo un proceso del sistema? y si es así ¿cómo se controla? Estas interesantes cuestiones han inspirado unas cuentas direcciones diferentes de diseño. La Figura 3.15 ilustra el rango de diferentes visiones que se encuentran en varios sistemas operativos contemporáneos.

Núcleo sin procesos. Una visión tradicional, que es común a muchos sistemas operativos antiguos, es la ejecución del sistema operativo fuera de todo proceso (Figura 3.15a). Con esta visión, cuando el proceso en ejecución se interrumpe o invoca una llamada al sistema, el contexto se guarda y el control pasa al núcleo. El sistema operativo tiene su propia región de memoria y su propia pila de sistema para controlar la llamada a procedimientos y sus retornos. El sistema operativo puede realizar todas las funciones que necesite y restaurar el contexto del proceso interrumpido, que hace que se retome la ejecución del proceso de usuario afectado. De forma alternativa, el sistema operativo puede realizar la salvaguarda del contexto y la activación de otro proceso diferente. Si esto ocurre o no depende de la causa de la interrupción y de las circunstancias puntuales en el momento.

En cualquier caso, el punto clave en este caso es que el concepto de proceso se aplica aquí únicamente a los programas de usuario. El código del sistema operativo se ejecuta como una entidad independiente que requiere un modo privilegiado de ejecución.

Ejecución dentro de los procesos de usuario. Una alternativa que es común en los sistemas operativos de máquinas pequeñas (PC, estaciones de trabajo) es ejecutar virtualmente todo el software de sistema operativo en el contexto de un proceso de usuario. Esta visión es tal que el sistema operativo se percibe como un conjunto de rutinas que el usuario invoca para realizar diferentes funciones, ejecutadas dentro del entorno del proceso de usuario. Esto se muestra en la Figura 3.15b. En un punto determinado, el sistema operativo maneja *n* imágenes de procesos. Cada imagen incluye no

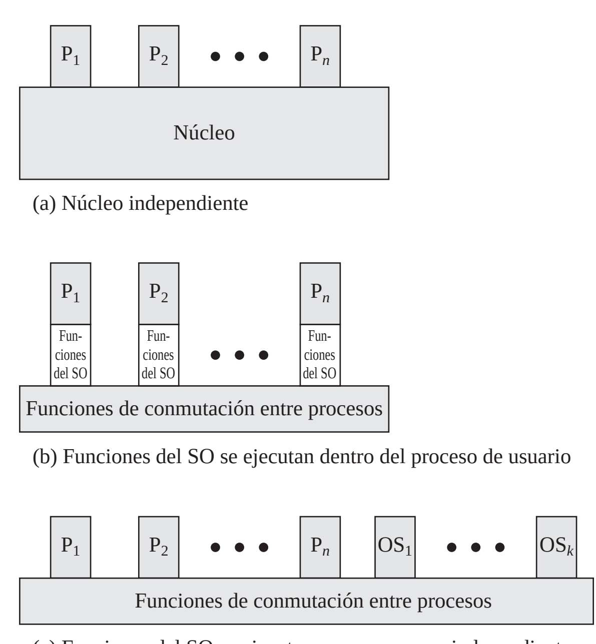


Figura 3.15. Relación entre el sistema operativo y los procesos de usuario.

sólo las regiones mostradas en la Figura 3.13, sino que además incluye áreas de programa, datos y pila para los programas del núcleo.

La Figura 3.16 sugiere la estructura de imagen de proceso típica para esta estrategia. Se usa una pila de núcleo separada para manejar llamadas/retornos cuando el proceso está en modo núcleo. El código de sistema operativo y sus datos están en el espacio de direcciones compartidas y se comparten entre todos los procesos.

Cuando ocurre una interrupción, *trap* o llamada al sistema, el procesador se pone en modo núcleo y el control se pasa al sistema operativo. Para este fin, el contexto se salva y se cambia de modo a una rutina del sistema operativo. Sin embargo, la ejecución continúa dentro del proceso de usuario actual. De esta forma, no se realiza un cambio de proceso, sino un cambio de modo dentro del mismo proceso.

Si el sistema operativo, después de haber realizado su trabajo, determina que el proceso actual debe continuar con su ejecución, entonces el cambio de modo continúa con el programa interrumpido, dentro del mismo proceso. Ésta es una de las principales ventajas de esta alternativa: se ha interrumpido un programa de usuario para utilizar alguna rutina del sistema operativo, y luego continúa, y todo esto se ha hecho sin incurrir en un doble cambio de proceso. Sin embargo, si se determina que se debe realizar un cambio de proceso en lugar de continuar con el proceso anterior, entonces el control pasa a la rutina de cambio de proceso. Esta rutina puede o no ejecutarse dentro del proceso actual, dependiendo del diseño del sistema. En algún momento, no obstante, el proceso actual se debe poner en un estado de no ejecución, designando a otro porceso como el siguiente a

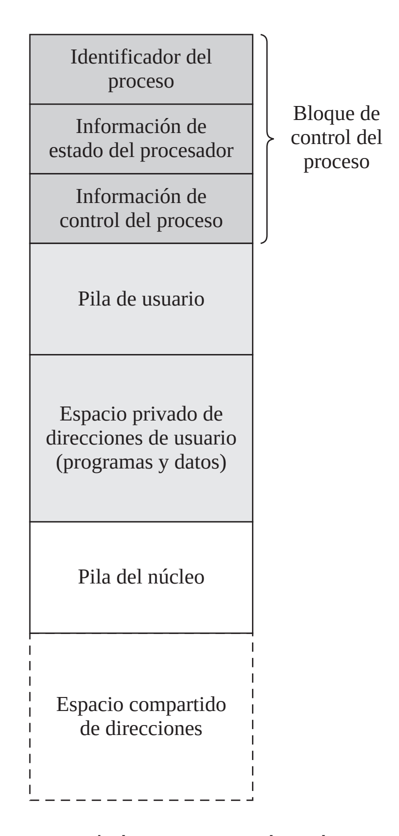


Figura 3.16. Imagen de proceso: el sistema operativo ejecuta dentro del espacio de usuario.

ejecutar. Durante esta fase, es más conveniente ver la ejecución como fuera de cualquiera de los procesos.

De alguna forma, esta visión de un sistema operativo es muy reseñable. De una forma más sencilla, en determinados instantes de tiempo, un proceso salva su estado, elige otro proceso a ejecutar entre aquellos que están listos, y delega el control a dicho proceso. La razón por la cual este esquema no se convierte en una situación arbitraria y caótica es que durante los instantes críticos el código que se está ejecutando es código de compartido de sistema operativo, no código de usuario. Debido al concepto de modo usuario y modo núcleo, el usuario no puede interferir con las rutinas del sistema operativo, incluso cuando se están ejecutando dentro del entorno del proceso del usuario. Esto nos recuerda que existe una clara distinción entre el concepto de proceso y programa y que la relación entre los dos no es uno a uno. Dentro de un proceso, pueden ejecutarse tanto un programas de usuario como programas del sistema operativo. Los programas de sistema operativo que se ejecutan en varios procesos de usuario son idénticos.

Sistemas operativos basados en procesos. Otra alternativa, mostrada en la Figura 3.15c, es implementar un sistema operativo como una colección de procesos de sistema. Como en las otras opciones, el software que es parte del núcleo se ejecuta en modo núcleo. En este caso, las principales funciones del núcleo se organizan como procesos independientes. De nuevo, debe haber una pequeña cantidad de código para intercambio de procesos que se ejecuta fuera de todos los procesos.

Esta visión tiene diferentes ventajas. Impone una disciplina de diseño de programas que refuerza el uso de sistemas operativos modulares con mínimas y claras interfaces entre los módulos. Adicionalmente, otras funciones del sistema operativo que no sean críticas están convenientemente separadas como otros procesos. Por ejemplo, hemos mencionado anteriormente un programa de monitorización que recoge niveles de utilización de diferentes recursos (procesador, memoria, canales) y el ratio de progreso de los procesos en el sistema. Debido a que este programa no realiza ningún servicio a ningún otro programa activo, sólo se puede invocar por el sistema operativo. Como proceso, esta función puede ejecutarse a un nivel de prioridad determinado, intercalándose con otros procesos bajo el control del activador. Por último, la implementación del sistema operativo como un grupo de procesos en entornos de multiprocesadores y multicomputadores, en los cuales determinados servicios del sistema operativo se pueden enviar a procesadores dedicados, incrementando el rendimiento.

3.5. UNIX SVR4 PROCESS MANAGEMENT

UNIX System V hace uso de una gestión de procesos simple pero muy potente que es fácilmente visible a nivel de usuario. UNIX sigue el modelo mostrado en la Figura 3.15b, en la cual la gran mayoría del sistema operativo se ejecuta dentro del entorno del proceso. De esta forma se necesitan dos modos, usuario y núcleo. UNIX utiliza también dos categorías de procesos, procesos de sistema y procesos de usuario. Los procesos del sistema ejecutan en modo núcleo y ejecuta código del sistema operativo para realizar tareas administrativas o funciones internas, como reserva de memoria o *swapping* de procesos. Los procesos de usuario operan en modo usuario para ejecutar programas y utilidades y en modo núcleo para ejecutar instrucciones que pertenecen al núcleo. Un proceso de usuario entra en modo núcleo por medio de la solicitud de la llamada al sistema, cuando se genera una de excepción (fallo) o cuando ocurre una interrupción.

ESTADOS DE LOS PROCESOS

En los sistemas operativos UNIX se utilizan un total de nueve estados de proceso; estos se encuentran recogidos en la Tabla 3.9 y el diagrama de transiciones se muestra en la Figura 3.17 (basada en figura

en [BACH86]). Esta figura es similar a la Figura 3.9b, con dos estados de procesos dormidos, propios de UNIX, correspondientes a dos estados de bloqueo. Las diferencias se pueden resumir en:

- UNIX utiliza dos estados Ejecutando que indican si el proceso está ejecutando en modo usuario o en modo núcleo.
- Se debe realizar la distinción entre dos estados: (Listo para Ejecutar, en Memoria) y (Expulsado). Estos son esencialmente el mismo estado, como indica la línea punteada que los une. La distinción se realiza para hacer énfasis en la forma mediante la cual se llega al estado de Expulsado. Cuando un proceso ejecuta en modo núcleo (como resultado de una llamada al sistema, interrupción de reloj o interrupción de E/S), requerirá un tiempo para que el sistema operativo complete su trabajo y esté listo para devolver el control al proceso de usuario. En este punto, el núcleo decide expulsar al proceso actual en favor de uno de los procesos listos de mayor prioridad. En este caso, el proceso actual se mueve al estado Expulsado. Sin embargo, por cuestiones de activación, estos procesos en el estado Expulsado y todos aquellos en los estados Listo para Ejecutar en Memoria, forman una única cola.

La expulsión sólo puede ocurrir cuando un proceso está a punto de moverse de modo núcleo a modo usuario. Mientras los procesos ejecutan al modo núcleo, no pueden ser expulsados. Esto hace que los sistemas UNIX no sean apropiados para procesamiento en tiempo real. Se proporcionará una explicación de los requisitos para procesamiento de tiempo-real en el Capítulo 10.

En UNIX existen dos procesos con un interés particular. El proceso 0 es un proceso especial que se crea cuando el sistema arranca; en realidad, es una estructura de datos predefinida que se carga en cuanto el ordenador arranca. Es el proceso *swapper*. Adicionalmente, el proceso 0 lanza al proceso 1, que se denomina proceso *init*; todos los procesos del sistema tienen al proceso 1 como antecesor. Cuando un nuevo usuario interactivo quiere entrar en el sistema, es el proceso 1 el que crea un proceso de usuario para él. Posteriormente, el proceso del usuario puede crear varios procesos hijos que conforman una estructura de árbol de procesos, de esta forma cualquier aplicación en particular puede consistir en un número de procesos relacionados entre sí.

Tabla 3.9. Estados de procesos en UNIX.

Ejecutando Usuario	Ejecutando en modo usuario.
Ejecutando Núcleo	Ejecutando en modo núcleo.
Listo para Ejecutar, en Memoria	Listo para ejecutar tan pronto como el núcleo lo planifique.
Dormido en Memoria	No puede ejecutar hasta que ocurra un evento; proceso en memoria principal (estado de bloqueo).
Listo para Ejecutar, en *Swap*	El proceso está listo para preguntar, pero el *swapper* debe cargar el proceso en memoria principal antes de que el núcleo pueda planificarlo para su ejecución.
Durmiendo, en *Swap*	El proceso está esperando un evento y ha sido expulsado al almacenamiento secundario (estado de bloqueo).
Expulsado	El proceso ha regresado de modo núcleo a modo usuario, pero el núcleo lo ha expulsado y ha realizado la activación de otro proceso.
Creado	El proceso ha sido creado recientemente y aún no está listo para ejecutar.
Zombie	El proceso ya no existe, pero deja un registro para que lo recoja su proceso padre.

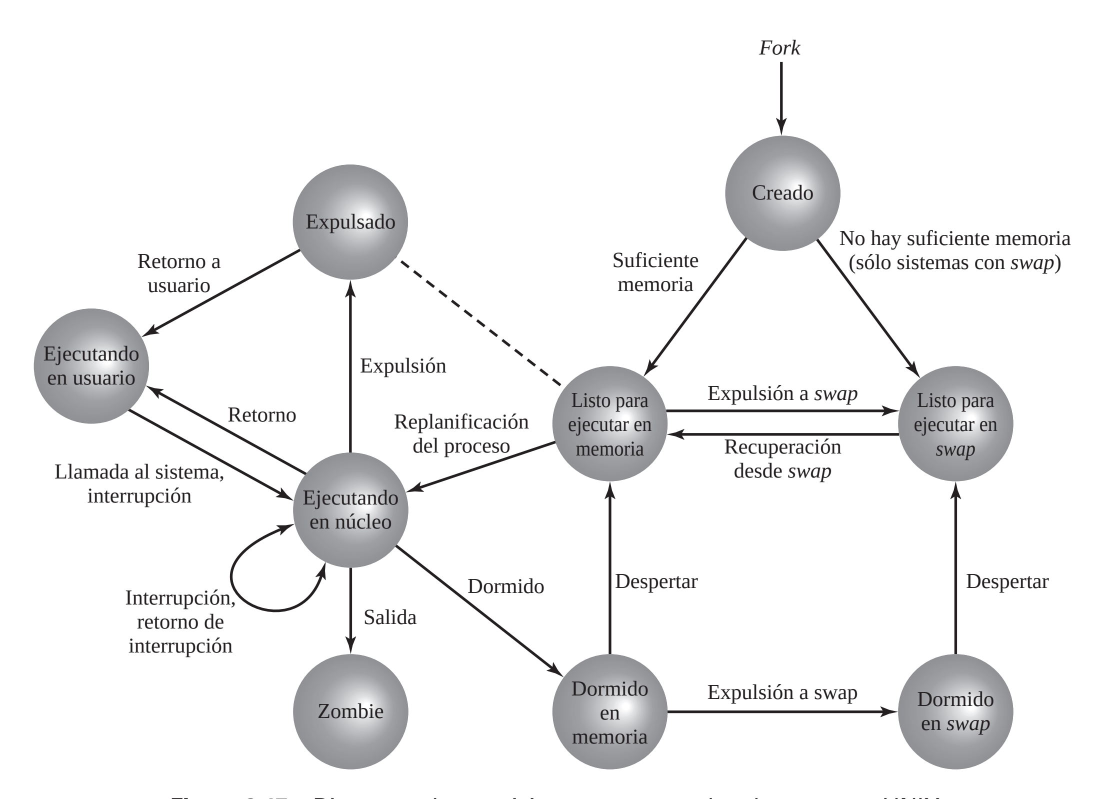


Figura 3.17. Diagrama de transiciones entre estados de procesos UNIX.

DESCRIPCIÓN DE PROCESOS

Un proceso UNIX es un conjunto de estructuras de datos, más bien complejas, que proporcionan al sistema operativo toda la información necesaria para manejar y activar los procesos. La Tabla 3.10 recoge los elementos de la imagen de proceso, que están organizados en tres partes: contexto a nivel de usuario, contexto de registros, y contexto a nivel de sistema.

El **contexto a nivel de usuario** contiene los elementos básicos de un programa de usuario que se pueden generar directamente desde un fichero objeto compilado. Un programa de usuario se divide en áreas de texto y datos; el área texto es de sólo-lectura y se crea con la intención de contener las instrucciones del programa. Cuando el proceso está en ejecución, el procesador utiliza el área de pila de usuario para gestionar las llamadas a procedimientos y sus retornos, así como los parámetros pasados. El área de memoria compartida es un área de datos que se comparte con otros procesos. Sólo existe una única copia física del área de memoria compartida, pero, por medio de la utilización de la memoria virtual, se presenta a cada uno de los procesos que comparten esta región de memoria común dentro de su espacio de dirección. Cuando un proceso está ejecutando, la información de estado de procesador se almacena en el área de **contexto de registros**.

El **contexto a nivel de sistema** contiene la información restante que necesita el sistema operativo para manejar el proceso. Consiste en una parte estática, de tamaño fijo y que permanece como parte del proceso a lo largo de todo su tiempo de vida, y una parte dinámica, que varía de tamaño a lo largo de la vida del proceso. La entrada de la tabla de procesos es un elemento de la parte estática y contiene información de control del proceso que es accesible por parte del núcleo

en todo momento; además, en un sistema de memoria virtual, todas las entradas en la tabla de procesos se mantienen en memoria principal. La Tabla 3.11 muestra los contenidos de la entrada de la tabla de procesos. El área de usuario, o área U, contiene información adicional de proceso que necesita el núcleo cuando está ejecutando en el contexto de este proceso; también se utiliza cuando el proceso se pagina desde/hacia memoria (*swapping*). La Tabla 3.12 muestra los contenidos de dicha tabla.

Tabla 3.10. Imagen de un proceso UNIX.

	Contexto a nivel de usuario
Texto	Instrucciones máquina ejecutables del programa.
Datos	Datos accesibles por parte del programa asociado a dicho proceso.
Pila de usuario	Contiene los argumentos, las variables locales, y los punteros a funciones ejecutadas en modo usuario.
Memoria compartida	Memoria compartida con otros procesos, usada para la comunicación entre procesos.
	Contexto de registros
Contador de programa	Dirección de la siguiente instrucción a ejecutar; puede tratarse del espacio de memoria del núcleo o de usuario de dicho proceso.
Registro de estado del procesador	Contiene el estado del hardware del procesador en el momento de la expulsión; los contenidos y el formato dependen específicamente del propio hardware.
Puntero de pila	Apunta a la cima de la pila de núcleo o usuario, dependiendo del modo de operación en el momento de la expulsión.
Registros de propósito general	Depende del hardware.
	Contexto nivel de sistema
Entrada en la tabla de procesos	Define el estado del proceso; esta información siempre está accesible por parte de sistema operativo.
Área U (de usuario)	Información de control del proceso que sólo se necesita acceder en el contexto del propio proceso.
Tabla de regiones por proceso	Define la traducción entre las direcciones virtuales y físicas; también contiene información sobre los permisos que indican el tipo de acceso permitido por parte del proceso; sólo-lectura, lectura-escritura, o lectura-ejecución.
Pila del núcleo	Contiene el marco de pila de los procedimientos del núcleo cuando el proceso ejecuta en modo núcleo.

Tabla 3.11. Entrada en la tabla de procesos UNIX.

Estado del proceso	Estado actual del proceso.
Punteros	Al área U y al área de memoria del proceso (texto, datos, pila).
Tamaño de proceso	Permite al sistema operativo conocer cuánto espacio está reservado para este proceso.
Identificadores de usuario	El **ID de usuario real** identifica el usuario que es responsable de la ejecución del proceso. El **ID de usuario efectivo** se puede utilizar para que el proceso gane, de forma temporal, los privilegios asociados a un programa en particular; mientras ese programa se ejecuta como parte del proceso, el proceso opera con el identificador de usuario efectivo.
Identificadores de proceso	Identificador de este proceso; identificador del proceso padre. Estos identificadores se fijan cuando el proceso entra en el estado Creado después de la llamaba al sistema *fork*.
Descriptor de evento	Válido cuando el proceso está en un estado dormido; cuando el evento ocurre, el proceso se mueve al estado Listo para Ejecutar.
Prioridad	Utilizado en la planificación del proceso.
Señal	Enumera las señales enviadas a este proceso pero que no han sido aún manejadas.
Temporizadores	Incluye el tiempo de ejecución del proceso, la utilización de recursos de núcleo, y el temporizador fijado por el usuario para enviar la señal de alarma al proceso.
P_link	Puntero al siguiente enlace en la cola de Listos (válido si proceso está Listo para Ejecutar).
Estado de memoria	Indica si la imagen del proceso se encuentra en memoria principal o secundaria. Si está en memoria, este campo indica si puede ser expulsado a *swap* o si está temporalmente fijado en memoria principal.

La distinción entre la entrada de la tabla de procesos y el área U refleja el hecho de que el núcleo de UNIX siempre ejecuta en el contexto de un proceso. La mayor parte del tiempo, el núcleo está realizando tareas relacionadas con dicho proceso. Sin embargo, otra parte del tiempo, como cuando el núcleo está realizando tareas de planificación preparatorias para la activación de otro proceso, se necesita la información sobre la totalidad de procesos del sistema. La información en la tabla de procesos es accesible cuando el proceso específico no es el que actualmente está en ejecución.

La tercera parte estática de contexto a nivel de sistema es una tabla de regiones por proceso, que se utiliza para el sistema de gestión de memoria. Por último, la pila del núcleo es una parte dinámica de contexto a nivel de sistema. Esta pila se usa cuando el proceso está ejecutando en modo núcleo y contiene la información que debe salvaguardarse y restaurarse en las llamadas a procedimientos o cuando ocurre una interrupción.

Tabla 3.12. Área U de UNIX.

Puntero a la tabla de proceso	Indica la entrada correspondiente a esta área U.
Identificador de usuario	Identificador de usuario real y efectivo. Utilizado para determinar los privilegios.
Temporizadores	Registro del tiempo que el proceso (y sus descendientes) han utilizado para ejecutar en modo usuario y modo núcleo.
Vector de manejadores de señales	Para cada tipo de señal definida en el sistema, se indica cómo el proceso va a reaccionar a la hora de recibirla (salir, ignorar, ejecutar una función específica definida por el usuario).
Terminal de control	Indica el terminal de acceso (*login*) para este proceso, si existe.
Campo de error	Recoge los errores encontrados durante una llamada al sistema.
Valor de retorno	Contiene los resultados de una llamada al sistema.
Parámetros de E/S	Indica la cantidad de datos transferidos, la dirección fuente (o destino) del vector de datos en el espacio de usuario, y los desplazamientos en fichero para la E/S.
Parámetros en fichero	Directorio actual y directorio raíz, dentro del sistema de ficheros, asociado al entorno de este proceso.
Tabla de descriptores de fichero de usuario	Recoge los ficheros del proceso abierto.
Campos límite	Restringe el tamaño del proceso y el tamaño máximo de fichero que puede crear.
Campos de los modos de permiso	Máscara de los modos de protección para la creación de ficheros por parte de este proceso.

CONTROL DE PROCESOS

La creación de procesos en UNIX se realiza por medio de la llamada al sistema *fork()*. Cuando con un proceso solicita una llamada *fork*, el sistema operativo realiza las siguientes funciones [BACH86]:

1. Solicita la entrada en la tabla de procesos para el nuevo proceso.
2. Asigna un identificador de proceso único al proceso hijo.
3. Hace una copia de la imagen del proceso padre, con excepción de las regiones de memoria compartidas.
4. Incrementa el contador de cualquier fichero en posesión del padre, para reflejar el proceso adicional que ahora también posee dichos ficheros.
5. Asigna al proceso hijo el estado Listo para Ejecutar.
6. Devuelve el identificador del proceso hijo al proceso padre, y un valor 0 al proceso hijo.

Todo este trabajo se realiza en modo núcleo, dentro del proceso padre. Cuando el núcleo ha completado estas funciones puede realizar cualquiera de las siguientes acciones, como parte de la rutina del activador:

1. Continuar con el proceso padre. El control vuelve a modo usuario en el punto en el que se realizó la llamada *fork* por parte del padre.
2. Transferir el control al proceso hijo. El proceso hijo comienza ejecutar en el mismo punto del código del padre, es decir en el punto de retorno de la llamada *fork*.
3. Transferir el control a otro proceso. Ambos procesos, padre e hijo, permanecen en el estado Listos para Ejecutar.

Puede resultar quizá un poco difícil visualizar este modo de creación de procesos debido a que ambos procesos, padre e hijo, están ejecutando el mismo segmento de código. La diferencia reside en que: cuando se retorna de la función *fork*, el parámetro de retorno se comprueba. Si el valor es 0, entonces este es el proceso hijo, y se puede realizar una bifurcación en la ejecución del programa para continuar con la ejecución de programa hijo. Si el valor no es 0, éste es el proceso padre, y puede continuar con la línea principal ejecución.

3.6. RESUMEN

El concepto fundamental dentro de los sistemas operativos modernos es el concepto de proceso. La función principal de un sistema operativo es crear, gestionar y finalizar los procesos. Cuando un proceso está activo, el sistema operativo debe ver cómo reservar tiempo para su ejecución por parte del procesador, coordinar sus actividades, gestionar las demandas que planteen conflictos, y reservar recursos del sistema para estos procesos.

Para realizar estas funciones de gestión de procesos, el sistema operativo mantiene una descripción de cada proceso o imagen de proceso, que incluye el espacio de direcciones dentro del cual el proceso está ejecutando, y el bloque de control de proceso. Este último contiene toda la información que el sistema operativo necesita para gestionar el proceso, incluyendo el estado actual, la reserva de recursos, la prioridad y otros datos de relevancia.

Durante su tiempo de vida, un proceso se mueve a lo largo de diferentes estados. Los más importantes de estos estados son Listo, Ejecutando, y Bloqueado. Un proceso Listo es un proceso que no está actualmente en ejecución pero que está listo para ser ejecutado tan pronto el sistema operativo lo active. Un proceso que está Ejecutando es aquel que está actualmente en ejecución por el procesador. En los sistemas multiprocesador, podrá haber varios procesos en este estado. Un proceso Bloqueado está esperando a que se complete un determinado evento, como una operación de E/S.

Un proceso en ejecución se interrumpe bien por una interrupción, que es un evento que ocurre fuera de proceso y que es recogido por el procesador, o por la ejecución de una llamada al sistema. En cualquier caso, el procesador realiza un cambio de modo, que es la transferencia de control a unas rutinas del sistema operativo. El sistema operativo, después de haber realizado el trabajo necesario, puede continuar con el proceso interrumpido o puede cambiar a otros procesos.

3.7. LECTURAS RECOMENDADAS

Todos los libros de texto listados de la Sección 2.9 cubren el material de este capítulo. Se pueden encontrar en [GOOD94] y [GRAY97] unas buenas descripciones de la gestión de procesos en UNIX. [NEHM75] es una descripción interesante de los estados de proceso del sistema operativo y las funciones necesarias para la activación de procesos.

GOOD94 Goodheart, B., y Cox, J. *The magic Garden Explained: The Internals of UNIX System V Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

GRAY97 Gray, J. *Interprocess Communication in Unix: The Nooks and Crannies.* Upper Saddler River, NJ: Prentice Hall, 1997.

NEHM75 Nehmer, J. «Dispatcher Primitives for the Construction of Operating System Kernels.» *Acta Informática*, vol 5, 1975.

3.8. TÉRMINOS CLAVE, CUESTIONES DE REPASO, Y PROBLEMAS

TÉRMINOS CLAVE

bloque de control de proceso	imagen de proceso	proceso hijo
cambio de modo	interrupción	proceso padre
cambio de proceso	modo núcleo	*round-robin*
estado bloqueado	modo privilegiado	*swapping*
estado ejecutando	modo sistema	tarea
estado listo	modo usuario	traza
estado saliente	nuevo estado	*trap*
estado suspendido	palabra de estado de programa	
expulsión	proceso	

CUESTIONES DE REPASO

3.1. ¿Qué es una traza de instrucciones?

3.2. ¿Cuáles son los eventos comunes que llevan a la creación de un proceso?

3.3. Para el modelo de procesamiento de la Figura 3.6, defina brevemente cada estado.

3.4. ¿Qué significa la expulsión de un proceso?

3.5. ¿Que es el *swapping* y cuál es su objetivo?

3.6. ¿Por qué la Figura 3.9 tiene dos estados bloqueados?

3.7. Indique cuatro características de un proceso suspendido.

3.8. ¿Para qué tipo de entidades el sistema operativo mantiene tablas de información por motivos de gestión?

3.9. Indique tres categorías generales de información que hay en el bloque de control de proceso.

3.10. ¿Por qué se necesitan dos modos (usuario y núcleo)?

3.11. ¿Cuáles son los pasos que realiza el sistema operativo para la creación de un proceso?

3.12. ¿Cuál es la diferencia entre interrupción y *trap*?

3.13. Dé tres ejemplos de interrupción.

3.14. ¿Cuál es la diferencia entre cambio de modo y cambio de proceso?

PROBLEMAS

3.1. Nombre cinco actividades principales del sistema operativo respecto a la gestión de procesos, y de forma breve describa por qué cada una es necesaria.

3.2. En [PINK89], se definen para los procesos los siguientes estados: ejecuta (ejecutando), activo (listo), bloqueado, y suspendido. Un proceso está bloqueado si está esperando el permiso para acceder a un recurso, y está bloqueado cuando está esperando a que se realice una operación sobre un recurso que ya ha adquirido. En muchos sistemas operativos, estos dos estados están agrupados en el estado de bloqueado, y el estado suspendido tiene el sentido usado en este capítulo. Compare las ventajas de ambas definiciones.

3.3. Para el modelo de siete estados de la Figura 3.9b, dibuje un diagrama de colas similar al de la Figura 3.8b.

3.4. Considerando el diagrama de transiciones de la Figura 3.9b. Suponga que le toca al sistema operativo activar un proceso y que hay procesos en el estado Listo y Listo/Suspendido, y que al menos uno de los procesos en estado Listo/Suspendido tiene mayor prioridad que todos los procesos Listos. Existen dos políticas extremas (1) siempre activar un proceso en estado Listo, para minimizar el efecto del swapping y (2) siempre dar preferencia a los procesos con mayor prioridad, incluso cuando eso implique hacer swapping, y dicho swapping no fuese necesario. Sugiera una política que encuentre un punto medio entre prioridad y rendimiento.

3.5. La Tabla 3.13 muestra los estados de proceso del sistema operativo VAX/VMS.

a) ¿Puede proporcionar una justificación para la existencia de tantos estados distintos de espera?

b) ¿Por qué los siguientes estados no tienen una versión residente y en *swap*: espera por fallo de página, espera por colisión de página, espera a un evento común, espera a liberación de página, y espera por recurso?

c) Dibuje un diagrama de transiciones de estado que indique la acción o suceso que causa dicha transición.

3.6. El sistema operativo VAX/VMS utiliza cuatro modos de acceso al procesador para facilitar la protección y compartición de recursos del sistema entre los procesos. El modo de acceso determina:

- **Privilegios de ejecución de instrucciones.** ¿Qué instrucciones puede ejecutar el procesador?
- **Privilegios de acceso a memoria.** ¿Qué posiciones de memoria pueden acceder las instrucciones actuales?

Los cuatros modos de acceso son:

- **Núcleo.** Ejecuta el núcleo del sistema operativo VMS, que incluye gestión de memoria, manejo de interrupciones, y operaciones de E/S.
- **Ejecutivo.** Ejecuta muchas de las llamadas al sistema, incluyendo las rutinas de gestión de ficheros y registros (disco o cinta).
- **Supervisor.** Ejecuta otros servicios del sistema operativo, tales como las respuestas a los mandatos del usuario.
- **Usuario.** Ejecuta los programas de usuario, además de las utilidades como compiladores, editores, enlazadores, y depuradores.

Tabla 3.13. VAX/VMS Estados del Proceso

Estado del Proceso	**Condición del Proceso**
Actualmente en ejecución	Proceso en ejecución.
Computable (residente)	Listo y residente en memoria principal.
Computable (en swap)	Listo, pero expulsado de memoria principal.
Espera por fallo de página	El proceso ha hecho referencia a una página que no está en memoria principal y debe esperar a que dicha página se lea.
Espera por colisión de página	El proceso ha hecho referencia a una página compartida que ha sido la causa de una espera por fallo de página en otro proceso, o una página privada que está en fase de ser leída en memoria o escrita a disco.
Espera a un evento común	Esperando a un *flag* de evento compartido (*flags* de evento son mecanismos de comunicación entre procesos por señales de un solo bit).
Espera por liberación de página	Esperando a que se libere una nueva página en memoria principal, que será asignada a un conjunto de páginas asignadas a dicho proceso (el conjunto de trabajo del proceso).
Espera hibernada (residente)	El proceso se ha puesto él mismo en estado de espera.
Espera hibernada (en swap)	Proceso hibernado que ha sido expulsado de memoria principal.
Espera a un evento local (residente)	El proceso está en memoria principal y a la espera de un *flag* de evento local (habitualmente a la espera de la finalización de una operación de E/S).
Espera a un evento local (en swap)	Proceso en espera de un evento local que ha sido expulsado de memoria principal.
Espera suspendida (residente)	El proceso se ha puesto en estado de espera por otro proceso.
Espera suspendida (en swap)	Proceso en espera suspendida que ha sido expulsado de memoria principal.
Espera de recurso	El proceso está esperando a disponer de un recurso de tipo general del sistema.

Un proceso que ejecuta en el modo con menos privilegios a menudo necesita llamar a un procedimiento que se ejecuta en un nivel de mayor privilegio; por ejemplo, un programa de usuario requiere una llamada al sistema. Esta llamada se realiza por medio de una instrucción de cambio de modo (CHM), que causa una interrupción que transfiere el control a una rutina en el nuevo modo de acceso. El retorno se realiza por medio de la instrucción REI (retorno de una excepción o interrupción).

a) Un gran número de sistemas operativos sólo tienen dos modos, núcleo y usuario. ¿Cuáles son las ventajas y los inconvenientes de tener cuatro modos en lugar de dos?

b) ¿Puede incluir un caso en el que haya más de cuatro modos?

3.7. El esquema de VMS comentado en el problema anterior se denomina habitualmente estructura de protección en anillo, tal y como se ilustra en la Figura 3.18. En realidad, el modelo sencillo núcleo/usuario, tal y como se describe en la Sección 3.3, es una estructura de dos anillos. [SILB04] aborda el problema con el siguiente enfoque.

La principal desventaja del modelo de estructura en anillo (jerárquico) es que no nos permite forzar el principio de necesidad-de-conocimiento. En particular, si un objeto debe ser

accesible en el dominio D_j pero no accesible en el dominio D_i, entonces debemos tener $j<i$. Pero esto implica que todo segmento accesible en D_i también es accesible en D_j.

a) Explique claramente cuál es el problema indicado en esta última cita.

b) Sugiera una forma mediante la cual un sistema operativo con estructura en anillo pueda resolver este problema.

3.8. La Figura 3.7b sugiere que un proceso sólo se puede encontrar en una cola de espera por un evento en el mismo instante.

a) ¿Es posible que un proceso esté esperando por más de un evento a la vez? Proporcione un ejemplo.

b) En dicho caso, ¿cómo modificaría la estructura de colas de la figura para dar soporte a esta nueva funcionalidad?

3.9. En gran número de los primeros ordenadores, una interrupción hacía que automáticamente los registros del procesador se guardasen en unas posiciones determinadas asociadas con esa señal de interrupción en particular, ¿bajo qué circunstancias esta es una buena técnica? Explique, por qué en general no es así.

3.10. En la Sección 3.4, se indica que UNIX no es apropiado para aplicaciones de tiempo real porque un proceso ejecutando en modo núcleo no puede ser expulsado. Elabore más el razonamiento.

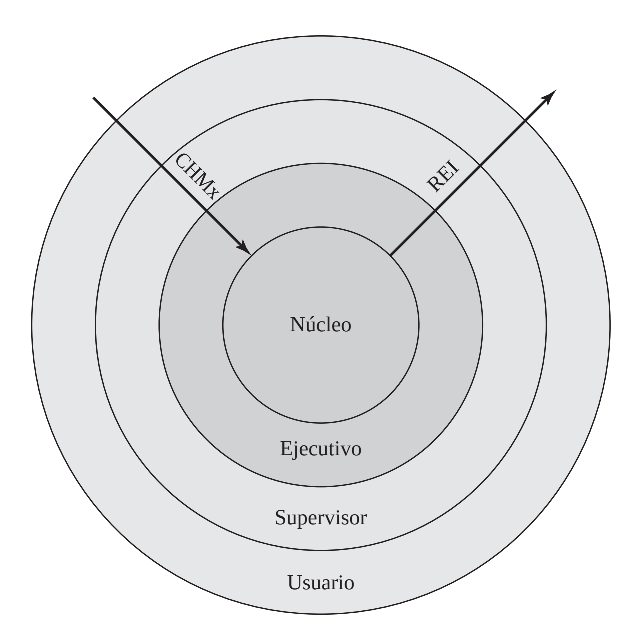


Figura 3.18. Modos de acceso de VAX/VMS.

PROYECTO DE PROGRAMACIÓN UNO. DESARROLLO DE UN INTÉRPRETE DE MANDATOS

El *shell* o intérprete de mandatos es la interfaz de usuario fundamental de los sistemas operativos. El primer proyecto es escribir un intérprete de mandatos sencillo – myshell – con las siguientes propiedades:

1. El intérprete de mandatos debe dar soporte a los siguientes mandatos internos:

   i. cd <directorio> – cambia el directorio actual a <directorio>. Si el argumento <directorio> no aparece, devuelve el directorio actual. Si el directorio no existe se debe proporcionar un mensaje de error apropiado. Este mandato debe modificar también la variable de entorno PWD.

   ii. clr – limpia la pantalla.

   iii. dir <directory> – lista el contenido de <directorio>.

   iv. environ – muestra todas las variables de entorno.

   v. echo <comentario> – muestra <comentario> en la pantalla seguido de una nueva línea (espacios múltiples o tabuladores se reducen a un especio sencillo).

   vi. help – muestra el manual de usuario usando el filtro more.

   vii. pause – detiene la ejecución del intérprete de mandatos hasta que se pulse 'Intro'.

   viii. quit – sale del intérprete de mandatos.

   ix. El entorno del intérprete de mandatos debe contener shell=<ruta>/myshell donde <ruta>/myshell es la ruta completa al ejecutable del intérprete de mandatos (no una ruta fijada al directorio inicial, sino la ruta real desde dónde se ha ejecutado).

2. Todo el resto de entradas por teclado se interpretan como la invocación de un programa, que deben realizarse por medio de un *fork* y la ejecución de dicho programa. Todo ello como un proceso hijo del intérprete de mandatos. Los programa deben ejecutarse en un entorno que incluya la entrada: parent=<ruta>/myshell donde <ruta>/myshell es la ruta descrita en el apartado 1.ix anterior.

3. El intérprete de mandatos debe ser capaz de leer su entrada de mandatos de un fichero. Por ejemplo, si se invoca al intérprete de mandatos con la línea:

   ```
   myshell fichero-lotes
   ```

   Donde fichero-lotes se supone que contiene las líneas de mandatos para el intérprete. Cuando se llegue al final del fichero, el intérprete de mandatos debe terminar. Obviamente, si el intérprete se invoca sin argumentos de entrada, solicitará los mandatos al usuario vía consola.

4. El intérprete de mandatos debe soportar redirección de E/S, sobre *stdin* y/o *stdout*. Por ejemplo, la línea de mandatos:

   ```
   nombreprograma arg1 arg2 < entrada > salida
   ```

   ejecutará el programa nombreprograma con los argumentos arg1 y arg2, el flujo de entra *stdin* se alimentará del fichero entrada y el flujo de salida *stdout* se volcará en el fichero salida.

   La redirección de *stdout* debe de ser posible también para los mandatos internos: dir, environ, echo, y help.

   Para la redirección de salida, si el carácter de redirección es > se creará el fichero salida si no existe y si existe se truncará su contenido. Si el indicador de redirección es >> se creará el fichero salida si no existe y si existe se añadirá la salida al final de su contenido.

5. El intérprete de mandatos debe soportar la ejecución de mandatos en segundo plano (*background*). Un signo & al final de la línea de mandatos indica que el intérprete debe devolver un *prompt* al usuario, inmediatamente después de haber lanzado el programa.

6. El *prompt* debe indicar la ruta del directorio actual.

Nota: se puede asumir que todos los argumentos en la línea de mandatos (incluyendo los símbolos de redirección `>`, `<`, y `>>`; y background `&`) estarán separados de los otros argumentos en la línea de mandatos por espacios en blanco – uno o más espacios y/o tabuladores (obsérvese la línea de mandatos en 4).

Requisitos del proyecto

1. Diseñe un intérprete de mandatos sencillo que satisfaga los criterios antes mencionados e impleméntelo en la plataforma UNIX seleccionada.
2. Escriba un manual sencillo que describa cómo usar el intérprete. El manual debe contener suficiente información para que un usuario principiante en UNIX pueda usarlo. Por ejemplo, se debe explicar los conceptos de redirección de E/S, de entorno de programa, y de ejecución en segundo plano (*background*). El manual **DEBE** llamarse `readme` y debe ser un documento de texto plano que pueda leerse con un editor de texto estándar.

   Como ejemplo del tipo de profundidad que se pide, deberá inspeccionar los manuales en línea de `csh` y `tcsh. (man csh, man tcsh)`. Estos intérpretes tienen mucha más funcionalidad que el que se pide, de forma que el manual que se requiere no deberá ser tan largo. No debe incluir bajo ningún concepto consideraciones de implementación, ficheros fuente o código. Esto se incluirá en otros ficheros del proyecto. Este manual debe ser un Manual de Usuario no un Manual de Desarrollo.
3. El código fuente **DEBE** estar extensamente comentado y apropiadamente estructurado, permitiendo a sus colegas comprenderlo y darle mantenimiento al código. ¡El código comentado con propiedad y bien alineado es mucho más fácil de interpretar e interesa que la persona que pueda evaluar su código pueda entenderlo con facilidad sin necesidad de hacer gimnasia mental!
4. Los detalles sobre el envío del proyecto se proporcionarán con antelación a la fecha límite.
5. El envío del proyecto debe contener sólo ficheros fuente, incluyendo ficheros de cabecera, `makefile` (en letras minúsculas, por favor) el fichero `readme` (en letras minúsculas, por favor). No se debe incluir ningún fichero ejecutable. El evaluador recompilará automáticamente su intérprete de mandatos a partir del código fuente. Si el código fuente no compila, no será calificado.
6. El `makefile` (en letras minúsculas, por favor) **DEBE** generar un fichero binario llamado `myshell` (en letras minúsculas, por favor). Un ejemplo de `makefile` sería:

   ```
   # Pepe Potamo, s1234567 – Proyecto 1 de SO
   # CompLab1/01 tutor: Chema Peña
   myshell: myshell.c utility.c myshell.h
   	gcc –Wall myshell.c utility.c –o myshell
   ```

   El programa `myshell` se generará simplemente tecleando `make` en la línea de mandatos.

Nota: la cuarta línea del `makefile` de ejemplo **DEBE** comenzar por un tabulador.

7. En el ejemplo mostrado arriba los ficheros incluidos en el directorio de envío eran:

   ```
   makefile
   myshell.c
   utility.c
   myshell.h
   readme
   ```

Envío

Es necesario un `makefile`. Todos los ficheros incluidos en su envío se copiarán al mismo directorio, por tanto no incluya rutas en su `makefile`. El `makefile` debe incluir todas las dependencias para compilar el programa. Si se incluye una biblioteca, su `makefile` debe construir dicha biblioteca.

Para dejar esto claro: ***no construya a mano ningún binario o fichero objeto***. Todo lo que se requerirá serán sus ficheros fuente, un `makefile`, y el fichero `readme`. Verifique su proyecto, copiando dichos ficheros a un directorio vacío y compilándolo completamente por medio del mandato `make`.

Usaremos un *script* que copia sus ficheros a un directorio de prueba, borre el fichero `myshell` previo, así como todos los ficheros `*.a`, y/o los `*.o`, y ejecute un `make`, copie una serie de ficheros de pruebas al directorio, y compruebe su intérprete de mandatos por medio de una serie de ficheros por lotes pasados por la entrada estándar (*stdin*) y por línea de mandatos. Si esta batería de pruebas falla debido a nombres erróneos, diferencias entre mayúsculas y minúsculas, versiones erróneas de código que fallen al compilar o falta de ficheros, etc., la secuencia de evaluación se detendrá. En este caso, la calificación obtenida será la de las pruebas pasadas completamente, la puntuación sobre el código fuente y el manual.

Documentación solicitada

En primer lugar, su código fuente se comprobará y evaluará así como el manual `readme`. Los comentarios son completamente necesarios en su código fuente. El manual de usuario se podrá presentar en el formato que se desee (siempre que se pueda visualizar por medio de un editor de texto estándar). Además, el manual debe contener suficiente información para que un usuario principiante de UNIX pueda usar el intérprete. Por ejemplo, se debe explicar los conceptos de redirección de E/S, de entorno de programa, y de ejecución en segundo plano (*background*). El manual **DEBE** llamarse `readme` (todo en minúsculas, por favor, **SIN** extensión `.txt`)

CAPÍTULO 4

Hilos, SMP y micronúcleos

Este capítulo analiza algunos conceptos avanzados relativos a la gestión de procesos que se pueden encontrar en los sistemas operativos modernos. En primer lugar, se muestra cómo el concepto de proceso es más complejo y sutil de lo que se ha visto hasta este momento y, de hecho, contiene dos conceptos diferentes y potencialmente independientes: uno relativo a la propiedad de recursos y otro relativo a la ejecución. En muchos sistemas operativos esta distinción ha llevado al desarrollo de estructuras conocidas como hilos (*threads*). Después de analizar los hilos se pasa a ver el multiprocesamiento simétrico (*Symmetric Multiprocessing*, SMP). Con SMP el sistema operativo debe ser capaz de planificar simultáneamente diferentes procesos en múltiples procesadores. Por último, se introduce el concepto de micronúcleo, una forma útil de estructurar el sistema operativo para dar soporte al manejo de procesos y sus restantes tareas.

4.1. PROCESOS E HILOS

Hasta este momento se ha presentado el concepto de un proceso como poseedor de dos características:

- **Propiedad de recursos.** Un proceso incluye un espacio de direcciones virtuales para el manejo de la imagen del proceso; como ya se explicó en el Capítulo 3 la imagen de un proceso es la colección de programa, datos, pila y atributos definidos en el bloque de control del proceso. De vez en cuando a un proceso se le puede asignar control o propiedad de recursos tales como la memoria principal, canales E/S, dispositivos E/S y archivos. El sistema operativo realiza la función de protección para evitar interferencias no deseadas entre procesos en relación con los recursos.
- **Planificación/ejecución.** La ejecución de un proceso sigue una ruta de ejecución (traza) a través de uno o más programas. Esta ejecución puede estar intercalada con ese u otros procesos. De esta manera, un proceso tiene un estado de ejecución (Ejecutando, Listo, etc.) y una prioridad de activación y ésta es la entidad que se planifica y activa por el sistema operativo.

En la mayor parte de los sistemas operativos tradicionales, estas dos características son, realmente, la esencia de un proceso. Sin embargo, debe quedar muy claro que estas dos características son independientes y podrían ser tratadas como tales por el sistema operativo. Así se hace en diversos sistemas operativos, sobre todo en los desarrollados recientemente. Para distinguir estas dos características, la unidad que se activa se suele denominar **hilo (*thread*)**, o **proceso ligero**, mientras que la unidad de propiedad de recursos se suele denominar **proceso** o **tarea**[1].

MULTIHILO

Multihilo se refiere a la capacidad de un sistema operativo de dar soporte a múltiples hilos de ejecución en un solo proceso. El enfoque tradicional de un solo hilo de ejecución por proceso, en el que no se identifica con el concepto de hilo, se conoce como estrategia monohilo. Las dos configuraciones que se muestran en la parte izquierda de la Figura 4.1 son estrategia monohilo. Un ejemplo de siste-

---

[1] Ni siquiera se puede mantener este grado de consistencia. En los sistemas operativos para *mainframe* de IBM, los conceptos de espacio de direcciones y tarea, respectivamente, más o menos se corresponden a los conceptos de proceso e hilo que se describen en esta sección. Además, en la literatura, el término *proceso ligero* se utiliza para (1) equivalente al término *hilo*, (2) un tipo particular de hilo conocido como hilo de nivel de núcleo, o (3) en el caso de Solaris, una entidad que asocia hilos de nivel de usuario con hilos de nivel de núcleo.

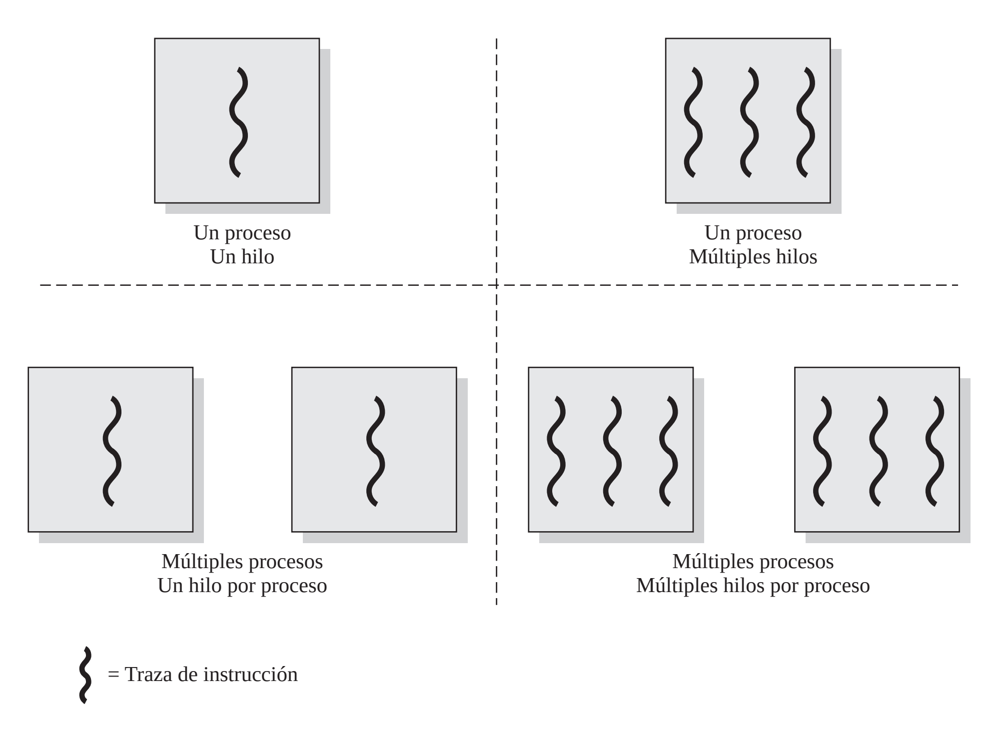


Figura 4.1. Hilos y procesos [ANDE97].

ma operativo que soporta un único proceso de usuario y un único hilo es el MS-DOS. Otros sistemas operativos, como algunas variedades de UNIX, soportan múltiples procesos de usuario, pero sólo un hilo por proceso. La parte derecha de la Figura 4.1 representa las estrategias multihilo. El entorno de ejecución de Java es un ejemplo de sistema con un único proceso y múltiples hilos. Lo interesante en esta sección es el uso de múltiples procesos, cada uno de los cuales soporta múltiples hilos. Este enfoque es el de Windows, Solaris, Mach, y OS/2 entre otros. En esta sección se ofrece una descripción general del mecanismo multihilo; más adelante en este capítulo se discuten los detalles de los enfoques de Windows, Solaris y Linux.

En un entorno multihilo, un proceso se define como la unidad de asignación de recursos y una unidad de protección. Se asocian con procesos los siguientes:

- Un espacio de direcciones virtuales que soporta la imagen del proceso.
- Acceso protegido a procesadores, otros procesos (para comunicación entre procesos), archivos y recursos de E/S (dispositivos y canales).

Dentro de un proceso puede haber uno o más hilos, cada uno con:

- Un estado de ejecución por hilo (Ejecutando, Listo, etc.).
- Un contexto de hilo que se almacena cuando no está en ejecución; una forma de ver a un hilo es como un contador de programa independiente dentro de un proceso.
- Una pila de ejecución.
- Por cada hilo, espacio de almacenamiento para variables locales.

- Acceso a la memoria y recursos de su proceso, compartido con todos los hilos de su mismo proceso.

La Figura 4.2 muestra la diferencia entre hilos y procesos desde el punto de vista de gestión de procesos. En un modelo de proceso monohilo (es decir, no existe el concepto de hilo), la representación de un proceso incluye su bloque de control de proceso y el espacio de direcciones de usuario, además de las pilas de usuario y núcleo para gestionar el comportamiento de las llamadas/retornos en la ejecución de los procesos. Mientras el proceso está ejecutando, los registros del procesador se controlan por ese proceso y, cuando el proceso no se está ejecutando, se almacena el contenido de estos registros. En un entorno multihilo, sigue habiendo un único bloque de control del proceso y un espacio de direcciones de usuario asociado al proceso, pero ahora hay varias pilas separadas para cada hilo, así como un bloque de control para cada hilo que contiene los valores de los registros, la prioridad, y otra información relativa al estado del hilo.

De esta forma, todos los hilos de un proceso comparten el estado y los recursos de ese proceso, residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo cambia determinados datos en memoria, otros hilos ven los resultados cuando acceden a estos datos. Si un hilo abre un archivo con permisos de lectura, los demás hilos del mismo proceso pueden también leer ese archivo.

Los mayores beneficios de los hilos provienen de las consecuencias del rendimiento:

1. Lleva mucho menos tiempo crear un nuevo hilo en un proceso existente que crear un proceso totalmente nuevo. Los estudios realizados por los que desarrollaron el sistema operativo Mach muestran que la creación de un hilo es diez veces más rápida que la creación de un proceso en UNIX [TEVA87].
2. Lleva menos tiempo finalizar un hilo que un proceso.
3. Lleva menos tiempo cambiar entre dos hilos dentro del mismo proceso.
4. Los hilos mejoran la eficiencia de la comunicación entre diferentes programas que están ejecutando. En la mayor parte de los sistemas operativos, la comunicación entre procesos inde-

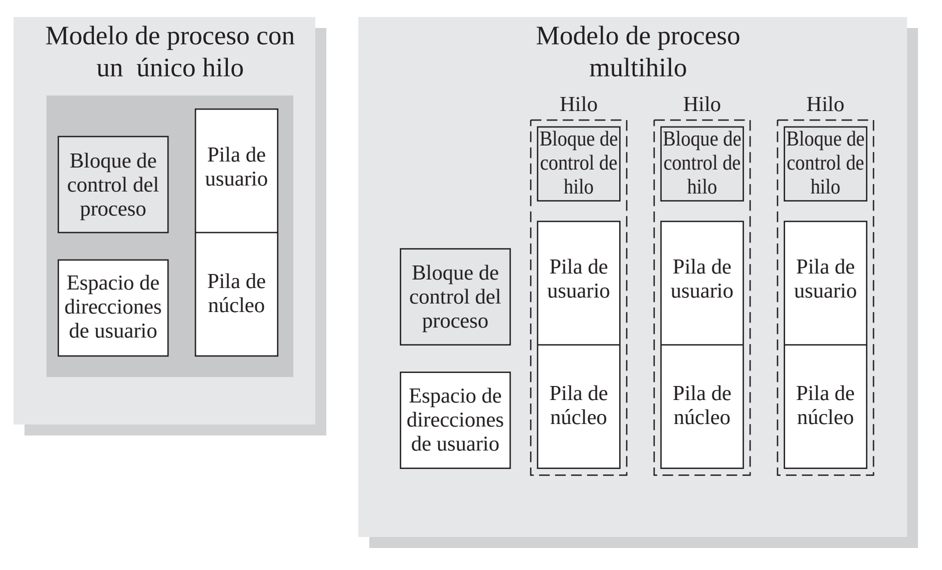


Figura 4.2. Modelos de proceso con un único hilo y multihilo.

pendientes requiere la intervención del núcleo para proporcionar protección y los mecanismos necesarios de comunicación. Sin embargo, ya que los hilos dentro de un mismo proceso comparten memoria y archivos, se pueden comunicar entre ellos sin necesidad de invocar al núcleo.

De esta forma, si se desea implementar una aplicación o función como un conjunto de unidades de ejecución relacionadas, es mucho más eficiente hacerlo con un conjunto de hilos que con un conjunto de procesos independientes.

Un ejemplo de una aplicación que podría hacer uso de hilos es un servidor de archivos. Cada vez que llega una nueva petición de archivo, el programa de gestión de archivos puede ejecutar un nuevo hilo. Ya que un servidor manejará muchas peticiones, se crearán y finalizarán muchos hilos en un corto periodo de tiempo. Si el servidor ejecuta en una máquina multiprocesador, pueden estar ejecutando simultáneamente múltiples hilos del mismo proceso en diferentes procesadores. Además, ya que los procesos o los hilos en un servidor de archivos deben compartir archivos de datos y, por tanto, coordinar sus acciones, es más rápido usar hilos y memoria compartida que usar procesos y paso de mensajes para esta coordinación.

A veces los hilos son también útiles en un solo procesador ya que ayudan a simplificar la estructura de programas que realizan varias funciones diferentes.

[LETW88] ofrece cuatro ejemplos de uso de hilos en un sistema de multiprocesamiento de un solo usuario:

- **Trabajo en primer plano y en segundo plano.** Por ejemplo, en un programa de hoja de cálculo, un hilo podría mostrar menús y leer la entrada de usuario, mientras otro hilo ejecuta los mandatos de usuario y actualiza la hoja de cálculo. Esta forma de trabajo a menudo incrementa la velocidad que se percibe de la aplicación, permitiendo al programa solicitar el siguiente mandato antes de que el mandato anterior esté completado.
- **Procesamiento asíncrono.** Los elementos asíncronos de un programa se pueden implementar como hilos. Por ejemplo, se puede diseñar un procesador de textos con protección contra un fallo de corriente que escriba el *buffer* de su memoria RAM a disco una vez por minuto. Se puede crear un hilo cuyo único trabajo sea crear una copia de seguridad periódicamente y que se planifique directamente a través del sistema operativo; no se necesita código adicional en el programa principal que proporcione control de tiempo o que coordine la entrada/salida.
- **Velocidad de ejecución.** Un proceso multihilo puede computar una serie de datos mientras que lee los siguientes de un dispositivo. En un sistema multiprocesador pueden estar ejecutando simultáneamente múltiples hilos de un mismo proceso. De esta forma, aunque un hilo pueda estar bloqueado por una operación de E/S mientras lee datos, otro hilo puede estar ejecutando.
- **Estructura modular de programas.** Los programas que realizan diversas tareas o que tienen varias fuentes y destinos de entrada y salida, se pueden diseñar e implementar más fácilmente usando hilos.

En un sistema operativo que soporte hilos, la planificación y la activación se realizan a nivel de hilo; de aquí que la mayor parte de la información de estado relativa a la ejecución se mantenga en estructuras de datos a nivel de hilo. Existen, sin embargo, diversas acciones que afectan a todos los hilos de un proceso y que el sistema operativo debe gestionar a nivel de proceso. Suspender un proceso implica expulsar el espacio de direcciones de un proceso de memoria principal para dejar hueco a otro espacio de direcciones de otro proceso. Ya que todos los hilos de un proceso comparten el mismo

espacio de direcciones, todos los hilos se suspenden al mismo tiempo. De forma similar, la finalización de un proceso finaliza todos los hilos de ese proceso.

FUNCIONALIDADES DE LOS HILOS

Los hilos, al igual que los procesos, tienen estados de ejecución y se pueden sincronizar entre ellos. A continuación se analizan estos dos aspectos de las funcionalidades de los hilos.

Estados de los hilos. Igual que con los procesos, los principales estados de los hilos son: Ejecutando, Listo y Bloqueado. Generalmente, no tiene sentido aplicar estados de suspensión a un hilo, ya que dichos estados son conceptos de nivel de proceso. En particular, si se expulsa un proceso, todos sus hilos se deben expulsar porque comparten el espacio de direcciones del proceso.

Hay cuatro operaciones básicas relacionadas con los hilos que están asociadas con un cambio de estado del hilo [ANDE97]:

- **Creación.** Cuando se crea un nuevo proceso, también se crea un hilo de dicho proceso. Posteriormente, un hilo del proceso puede crear otro hilo dentro del mismo proceso, proporcionando un puntero a las instrucciones y los argumentos para el nuevo hilo. Al nuevo hilo se le proporciona su propio registro de contexto y espacio de pila y se coloca en la cola de Listos.
- **Bloqueo.** Cuando un hilo necesita esperar por un evento se bloquea, almacenando los registros de usuario, contador de programa y punteros de pila. El procesador puede pasar a ejecutar otro hilo en estado Listo, dentro del mismo proceso o en otro diferente.
- **Desbloqueo.** Cuando sucede el evento por el que el hilo está bloqueado, el hilo se pasa a la cola de Listos.
- **Finalización.** Cuando se completa un hilo, se liberan su registro de contexto y pilas.

Un aspecto importante es si el bloqueo de un hilo implica el bloqueo del proceso completo. En otras palabras, si se bloquea un hilo de un proceso, ¿esto impide la ejecución de otro hilo del mismo proceso incluso si el otro hilo está en estado de Listo? Sin lugar a dudas, se pierde algo de la potencia y flexibilidad de los hilos si el hilo bloqueado bloquea al proceso entero.

Volveremos a este tema a continuación cuando veamos los hilos a nivel de usuario y a nivel de núcleo, pero por el momento consideraremos los beneficios de rendimiento de los hilos que no bloquean al proceso completo. La Figura 4.3 (basada en una de [KLEI96]) muestra un programa que realiza dos llamadas a procedimiento remoto (RPC)[2] a dos máquinas diferentes para poder combinar los resultados. En un programa de un solo hilo, los resultados se obtienen en secuencia, por lo que el programa tiene que esperar a la respuesta de cada servidor por turnos. Reescribir el programa para utilizar un hilo diferente para cada RPC, mejora sustancialmente la velocidad. Observar que si el programa ejecuta en un uniprocesador, las peticiones se deben generar en secuencia y los resultados se deben procesar en secuencia; sin embargo, el programa espera concurrentemente las dos respuestas.

En un uniprocesador, la multiprogramación permite el intercalado de múltiples hilos con múltiples procesos. En el ejemplo de la Figura 4.4, se intercalan tres hilos de dos procesos en un procesa-

---

[2] RPC es una técnica por la que dos programas, que pueden ejecutar en diferentes máquinas, interactúan utilizando la sintaxis y la semántica de las llamadas a procedimiento. Tanto el programa llamante como el llamado se comportan como si el otro programa estuviera ejecutando en la misma máquina. Los RPC se suelen utilizar en las aplicaciones cliente/servidor y se analizan en el Capítulo 13.

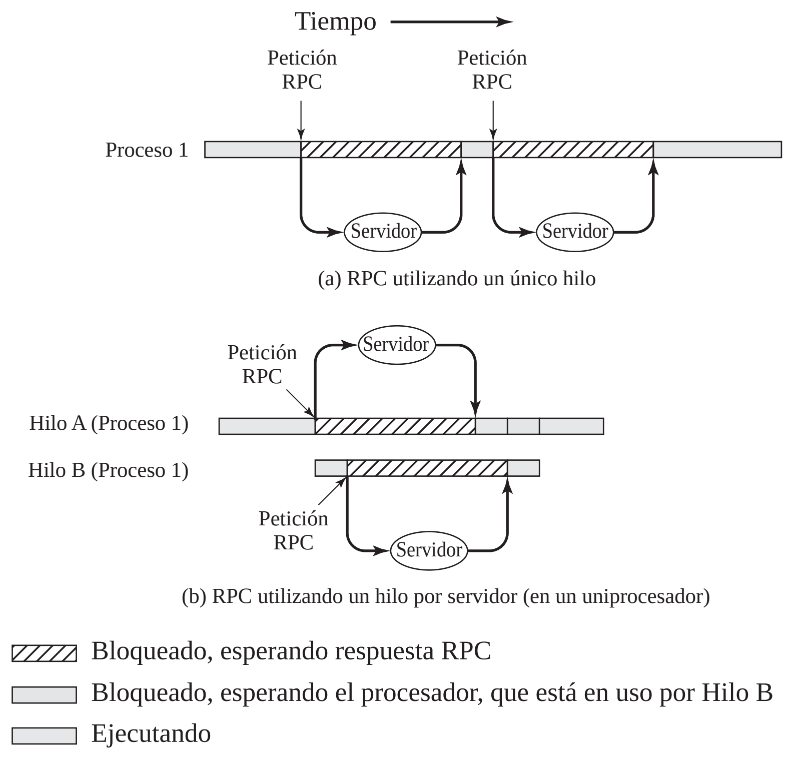


Figura 4.3. Llamadas a Procedimiento Remoto (RPC) utilizando hilos.

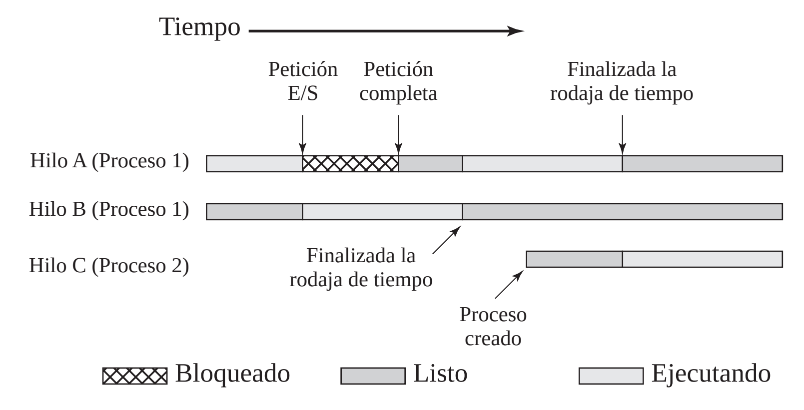


Figura 4.4. Ejemplo multihilo en un uniprocesador.

dor. La ejecución pasa de un hilo a otro cuando se bloquea el hilo actualmente en ejecución o su porción de tiempo se agota[3].

[3] En este ejemplo, el hilo C comienza a ejecutar después de que el hilo A finaliza su rodaja de tiempo, aunque el hilo B esté también listo para ejecutar. La elección entre B y C es una decisión de planificación, un tema que se aborda en la Parte Cuatro.

Sincronización de hilos. Todos los hilos de un proceso comparten el mismo espacio de direcciones y otros recursos, como por ejemplo, los archivos abiertos. Cualquier alteración de un recurso por cualquiera de los hilos, afecta al entorno del resto de los hilos del mismo proceso. Por tanto, es necesario sincronizar las actividades de los hilos para que no interfieran entre ellos o corrompan estructuras de datos. Por ejemplo, si dos hilos de modo simultáneo intentan añadir un elemento a una lista doblemente enlazada, se podría perder un elemento o la lista podría acabar malformada.

Los asuntos que surgen y las técnicas que se utilizan en la sincronización de los hilos son, en general, los mismos que en la sincronización de procesos. Estos aspectos y técnicas se tratan en los Capítulos 5 y 6.

EJEMPLO—ADOBE PAGEMAKER

Un ejemplo del uso de los hilos es la aplicación Adobe PageMaker cuando ejecuta en un sistema compartido. PageMaker es una herramienta de escritura, diseño y producción para entornos de escritorio. La estructura de hilos de PageMaker utilizada en OS/2, que se muestran la Figura 4.5 [KRON90], se eligió para optimizar la respuesta de la aplicación (se pueden encontrar estructuras de hilos similares en otros sistemas operativos). Siempre hay tres hilos activos: un hilo para manejar eventos, un hilo para repintar la pantalla y un hilo de servicio.

Generalmente, OS/2 es menos sensible en la gestión de ventanas si cualquier mensaje de entrada requiere demasiado proceso. En OS/2 se recomienda que ningún mensaje requiera más de 0,1 segundos de tiempo de procesamiento. Por ejemplo, llamar a una subrutina para imprimir una página mientras se procesa un mandato de impresión podría impedir al sistema el envío de más mensajes a

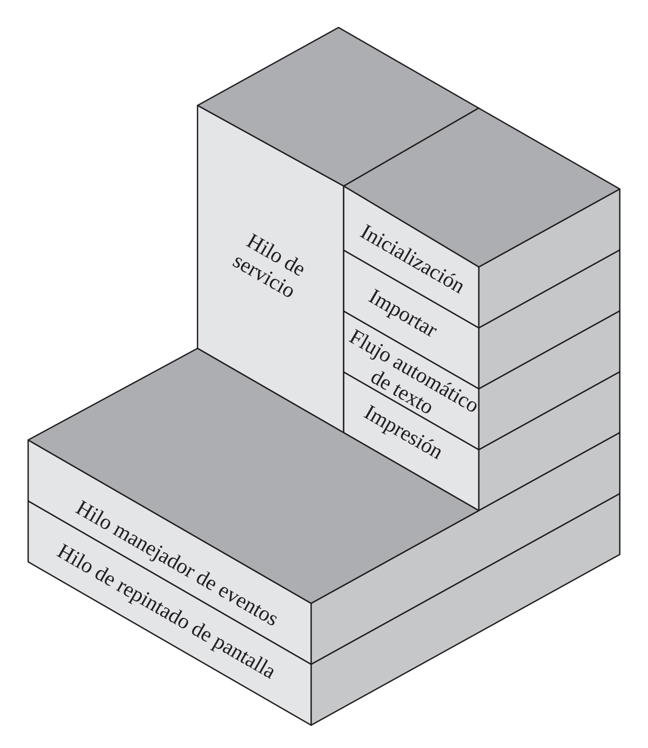


Figura 4.5. Estructura de hilos para Adobe PageMaker.

otras aplicaciones, degradando el rendimiento. Para cumplir este criterio, las operaciones de usuario que requieren mucho tiempo —imprimir, importar datos, y descargar texto— son realizadas por un hilo de servicio. En su mayor parte, la inicialización del programa también se lleva a cabo por el hilo de servicio, y utiliza el tiempo inactivo mientras el usuario invoca el diálogo de creación de un nuevo documento o abre un documento existente. Un hilo independiente espera nuevos mensajes de evento.

La sincronización del hilo de servicio y del hilo que gestiona los eventos es complicada porque un usuario puede continuar escribiendo o moviendo el ratón (lo cual activa el hilo de manejo de eventos), mientras el hilo de servicio está todavía ocupado. Si sucede este conflicto, PageMaker filtra estos mensajes y sólo acepta algunos básicos, tales como el redimensionamiento de la ventana.

El hilo de servicio manda un mensaje al hilo de manejo de eventos para indicar que ha finalizado su tarea. Hasta que esto ocurre, se restringe la actividad de usuario en PageMaker. El programa indica esta situación deshabilitando elementos de los menús y mostrando un cursor «ocupado». El usuario tiene la libertad de cambiar a otra aplicación; cuando el cursor se mueve a la nueva ventana, cambiará al cursor apropiado para esa aplicación.

La función de repintado de pantalla se maneja por un hilo diferente. Se realiza así por dos razones:

1. PageMaker no limita el número de objetos que aparecen en una página; de esta forma, procesar una petición de repintado puede exceder fácilmente la recomendación de 0,1 segundos.
2. Utilizar un hilo independiente permite al usuario cancelar el repintado. En este caso, cuando el usuario reescala una página, el repintado se puede realizar de forma inmediata. El programa es menos sensible si completa una visualización antigua antes de comenzar con una visualización en la nueva escala.

También es posible el desplazamiento dinámico línea a línea —repintar la pantalla a medida que el usuario mueve el indicador de desplazamiento—. El hilo de manejo de eventos monitoriza la barra de desplazamiento y repinta las reglas de los márgenes (haciéndolo de forma rápida para dar al usuario la posición actual de forma inmediata). Mientras tanto, el hilo de repintado de pantalla está constantemente intentando repintar la página.

La implementación del repintado dinámico sin el uso de múltiples hilos genera una gran sobrecarga en la aplicación. Múltiples hilos permiten separar actividades concurrentes de forma más natural en el código.

HILOS DE NIVEL DE USUARIO Y DE NIVEL DE NÚCLEO

Existen dos amplias categorías de implementación de hilos: hilos de nivel de usuario (*user-level threads*, ULT) e hilos de nivel de núcleo (*kernel-level threads*, KLT)[4]. Los últimos son también conocidos en la literatura como hilos soportados por el núcleo (*kernel-supported threads*) o procesos ligeros (*lightweight processes*).

Hilos de nivel de usuario. En un entorno ULT puro, la aplicación gestiona todo el trabajo de los hilos y el núcleo no es consciente de la existencia de los mismos. La Figura 4.6a muestra el enfoque

[4] Los acrónimos ULT y KLT son exclusivos de este libro y se introducen por concisión.

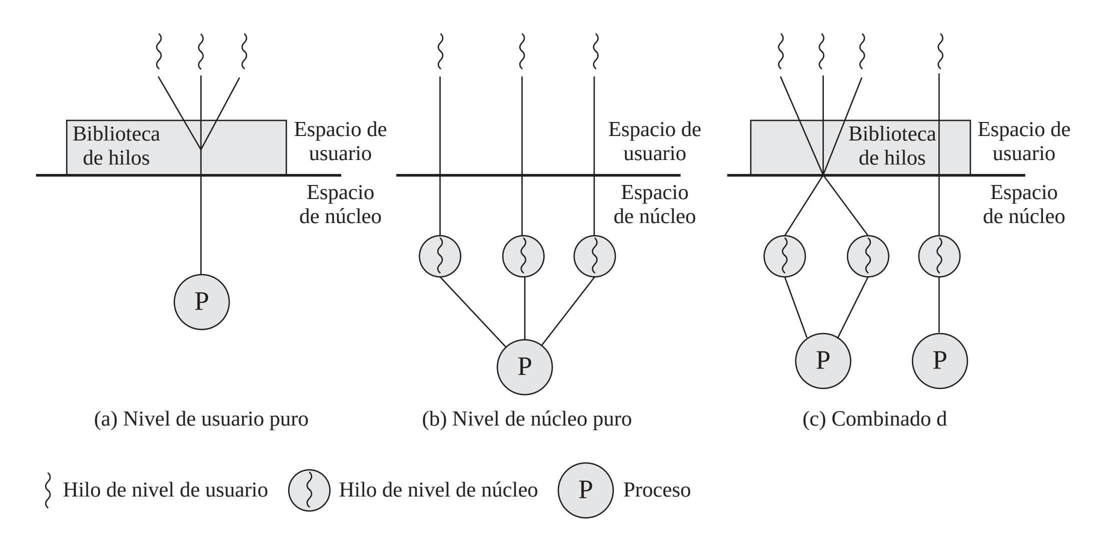


Figura 4.6. Hilos de nivel de usuario y de nivel de núcleo.

ULT. Cualquier aplicación puede programarse para ser multihilo a través del uso de una biblioteca de hilos, que es un paquete de rutinas para la gestión de ULT. La biblioteca de hilos contiene código para la creación y destrucción de hilos, para paso de mensajes y datos entre los hilos, para planificar la ejecución de los hilos, y para guardar y restaurar el contexto de los hilos.

Por defecto, una aplicación comienza con un solo hilo y ejecutando en ese hilo. Esta aplicación y su hilo se localizan en un solo proceso gestionado por el núcleo. En cualquier momento que la aplicación esté ejecutando (el proceso está en estado Ejecutando), la aplicación puede crear un nuevo hilo a ejecutar dentro del mismo proceso. La creación se realiza llamando a la utilidad de creación en la biblioteca de hilos. Se pasa el control a esta utilidad a través de una llamada a procedimiento. La biblioteca de hilos crea una estructura de datos para el nuevo hilo y luego pasa el control a uno de los hilos de ese proceso que esté en estado listo, utilizando algún algoritmo de planificación. Cuando se pasa el control a la biblioteca, se almacena el contexto del hilo actual, y cuando se pasa el control de la biblioteca al hilo, se recupera el contexto de ese hilo. El contexto tiene esencialmente el contenido de los registros del usuario, el contador que programa, y los punteros de pila.

Toda la actividad descrita en el párrafo anterior tiene lugar en el espacio de usuario y dentro de un solo proceso. El núcleo no es consciente de esta actividad. El núcleo continúa planificando el proceso como una unidad y asigna al proceso un único estado (Listo, Ejecutando, Bloqueado, etc.). Los siguientes ejemplos deberían aclarar la relación entre planificación de hilos y planificación de procesos. Suponer que el proceso B está ejecutando en su hilo 2; en la Figura 4.7a se muestran los estados del proceso y de dos ULT que son parte del proceso. Cada una de las siguientes es una posible situación:

1. La aplicación ejecutando en el hilo 2 hace una llamada al sistema que bloquea a B. Por ejemplo, se realiza una llamada de E/S. Esto hace que se pase el control al núcleo. El núcleo llama a la acción de E/S, sitúa al proceso B en estado de Bloqueado, y cambia a otro proceso. Mientras tanto, de acuerdo a la estructura de datos conservada en la biblioteca de los hilos, el hilo 2 del proceso B está todavía en estado Ejecutando. Es importante darse cuenta de que el hilo 2 no está ejecutando realmente en el sentido de estar corriendo en el procesador; pero se percibe como estado Ejecutando en la biblioteca de los hilos. Los diagramas de estado correspondientes se muestran en la Figura 4.7b.

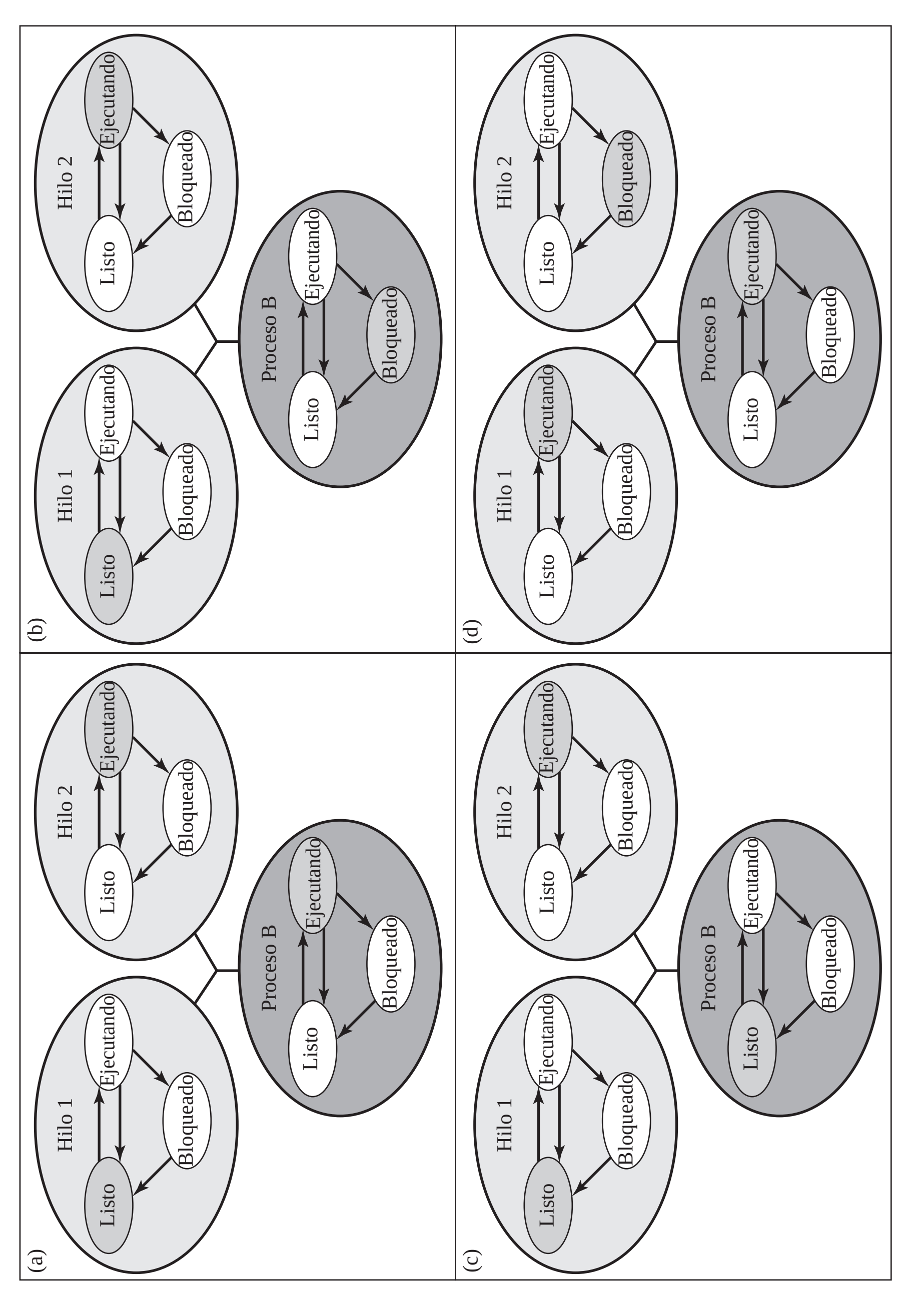


Figura 4.7. Ejemplos de relaciones entre los estados de los hilos de nivel de usuario y los estados de proceso.

2. Una interrupción de reloj pasa el control al núcleo y decide que el proceso actual en ejecución (B) ha finalizado su porción de tiempo. El núcleo pasa al proceso B a estado de Listo y cambia a otro proceso. Mientras tanto, de acuerdo a la estructura de datos conservada en la biblioteca de los hilos, el hilo 2 del proceso B está todavía en estado de Ejecución. Los diagramas de estado correspondientes se muestran en la Figura 4.7c.
3. El hilo 2 llega a un punto donde necesita una acción del hilo 1 o del proceso B. El hilo 2 entra en estado de Bloqueado y el hilo 1 pasa de Listo a Ejecutando. El proceso en sí continúa en estado Ejecutando. Los diagramas de estado correspondientes se muestran en la Figura 4.7d.

En los casos 1 y 2 (Figuras 4.7b y 4.7c), cuando el núcleo devuelve el control al proceso B, la ejecución se reanuda en el hilo 2. También hay que advertir que un proceso puede interrumpirse, bien por finalizar su porción de tiempo o bien por ser expulsado por un proceso de mayor prioridad, mientras está ejecutando código de la biblioteca de los hilos. De esta forma, un proceso puede estar en medio de una transición de un hilo a otro hilo cuando se interrumpe. Cuando se reanuda el proceso, la ejecución continúa con la biblioteca de los hilos, que completa el cambio de hilo y pasa el control al nuevo hilo del proceso.

El uso de ULT en lugar de KLT, presenta las siguientes ventajas:

1. El cambio de hilo no requiere privilegios de modo núcleo porque todas las estructuras de datos de gestión de hilos están en el espacio de direcciones de usuario de un solo proceso. Por consiguiente, el proceso no cambia a modo núcleo para realizar la gestión de hilos. Esto ahorra la sobrecarga de dos cambios de modo (usuario a núcleo; núcleo a usuario).
2. La planificación puede especificarse por parte de la aplicación. Una aplicación se puede beneficiar de un simple algoritmo de planificación cíclico, mientras que otra se podría beneficiar de un algoritmo de planificación basado en prioridades. El algoritmo de planificación se puede hacer a medida sin tocar el planificador del sistema operativo.
3. Los ULT pueden ejecutar en cualquier sistema operativo. No se necesita ningún cambio en el nuevo núcleo para dar soporte a los ULT. La biblioteca de los hilos es un conjunto de utilidades a nivel de aplicación que comparten todas las aplicaciones.

Hay dos desventajas de los ULT en comparación con los KLT:

1. En un sistema operativo típico muchas llamadas al sistema son bloqueantes. Como resultado, cuando un ULT realiza una llamada al sistema, no sólo se bloquea ese hilo, sino que se bloquean todos los hilos del proceso.
2. En una estrategia pura ULT, una aplicación multihilo no puede sacar ventaja del multiproceso. El núcleo asigna el proceso a un solo procesador al mismo tiempo. Por consiguiente, en un determinado momento sólo puede ejecutar un hilo del proceso. En efecto, tenemos multiprogramación a nivel de aplicación con un solo proceso. Aunque esta multiprogramación puede dar lugar a una mejora significativa de la velocidad de la aplicación, hay aplicaciones que se podrían beneficiar de la habilidad de ejecutar porciones de código de forma concurrente.

Hay formas de afrontar estos dos problemas. Por ejemplo, ambos problemas pueden superarse escribiendo una aplicación de múltiples procesos en lugar de múltiples hilos. Pero este enfoque elimina la principal ventaja de los hilos: cada cambio es un cambio de proceso en lugar de un cambio de hilo, lo que genera una gran sobrecarga.

Otra forma de solucionar el problema de hilos que se bloquean es una técnica denominada *jacketing* (revestimiento). El objetivo de esta técnica es convertir una llamada al sistema bloqueante en una

llamada al sistema no bloqueante. Por ejemplo, en lugar de llamar directamente a una rutina del sistema de E/S, un hilo puede llamar a una rutina *jacket* de E/S a nivel de aplicación. Con esta rutina *jacket*, el código verifica si el dispositivo de E/S está ocupado. Si lo está, el hilo entra en estado Bloqueado y pasa el control (a través de la biblioteca de hilos) a otro hilo. Cuando este hilo recupera de nuevo el control, chequea de nuevo el dispositivo de E/S.

Hilos a nivel de núcleo. En un entorno KLT puro, el núcleo gestiona todo el trabajo de gestión de hilos. No hay código de gestión de hilos en la aplicación, solamente una interfaz de programación de aplicación (API) para acceder a las utilidades de hilos del núcleo. Windows es un ejemplo de este enfoque.

La Figura 4.6b representa el entorno KLT puro. Cualquier aplicación puede programarse para ser multihilo. Todos los hilos de una aplicación se mantienen en un solo proceso. El núcleo mantiene información de contexto del proceso como una entidad y de los hilos individuales del proceso. La planificación realizada por el núcleo se realiza a nivel de hilo. Este enfoque resuelve los dos principales inconvenientes del enfoque ULT. Primero, el núcleo puede planificar simultáneamente múltiples hilos de un solo proceso en múltiples procesadores. Segundo, si se bloquea un hilo de un proceso, el núcleo puede planificar otro hilo del mismo proceso. Otra ventaja del enfoque KLT es que las rutinas del núcleo pueden ser en sí mismas multihilo.

La principal desventaja del enfoque KLT en comparación con el enfoque ULT es que la transferencia de control de un hilo a otro del mismo proceso requiere un cambio de modo al núcleo. Para mostrar estas diferencias, la Tabla 4.1 muestra los resultados de las medidas tomadas en un uniprocesador VAX ejecutando un sistema operativo tipo UNIX. Las dos medidas son las siguientes: *Crear un Proceso Nulo*, el tiempo para crear, planificar, ejecutar, y completar un proceso/hilo que llama al procedimiento nulo (es decir, la sobrecarga de crear un proceso/hilo); y *Señalizar-Esperar,* el tiempo que le lleva a un proceso/hilo señalizar a un proceso/hilo que está esperando y a continuación esperar una condición (es decir, la sobrecarga de sincronizar dos procesos/hilos).

Tabla 4.1. Latencia de las Operaciones en Hilos y Procesos (μS) [ANDE92].

Operación	Hilos a nivel de usuario	Hilos a nivel de núcleo	Procesos
Crear Proceso Nulo	34	948	11.300
Señalizar-Esperar	37	441	1.840

Como se puede apreciar puede haber más de un orden de diferencia entre ULT y KLT y de forma similar entre KLT y procesos.

De esta forma, mientras que hay una ganancia significativa entre el uso de multihilos KLT en comparación con procesos de un solo hilo, hay una ganancia significativa adicional por el uso de ULT. Sin embargo, depende de la naturaleza de la aplicación involucrada que nos podamos beneficiar o no de la ganancia adicional. Si la mayor parte de los cambios de hilo en una aplicación requieren acceso al modo núcleo, el esquema basado en ULT no sería tan superior al esquema basado en KLT.

Enfoques combinados. Algunos sistemas operativos proporcionan utilidades combinadas ULT/KLT (Figura 4.6c). Solaris es el principal ejemplo de esto. En un sistema combinado, la creación de hilos se realiza por completo en el espacio de usuario, como la mayor parte de la planificación y sincronización de hilos dentro de una aplicación. Los múltiples ULT de una aplicación se asocian en un número (menor o igual) de KLT. El programador debe ajustar el número de KLT para una máquina y aplicación en particular para lograr los mejores resultados posibles.

En los enfoques combinados, múltiples hilos de la misma aplicación pueden ejecutar en paralelo en múltiples procesadores, y una llamada al sistema bloqueante no necesita bloquear el proceso completo. Si el sistema está bien diseñado, este enfoque debería combinar las ventajas de los enfoques puros ULT y KLT, minimizando las desventajas.

OTRAS CONFIGURACIONES

Como hemos comentado, los conceptos de asignación de recursos y unidades de activación han sido tradicionalmente relacionados con el concepto de proceso; esto es, como una relación 1:1 entre hilos y procesos. Recientemente, ha habido mucho interés en proporcionar múltiples hilos dentro de un solo proceso, lo que es una relación muchos-a-uno. Sin embargo, como muestra la Tabla 4.2, las otras dos combinaciones han sido también investigadas, y se denominan relación muchos-a-muchos y relación uno-a-muchos.

Tabla 4.2. Relación Entre Hilos y Procesos.

Hilos:Procesos	Descripción	Sistemas de Ejemplo
1:1	Cada hilo de ejecución es un único proceso con su propio espacio de direcciones y recursos.	Implementaciones UNIX tradicionales.
M:1	Un proceso define un espacio de direcciones y pertenencia dinámica de recursos. Se pueden crear y ejecutar múltiples hilos en ese proceso.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH.
1:M	Un hilo puede migrar de un entorno de proceso a otro. Esto permite a los hilos moverse fácilmente entre distintos sistemas.	Ra (Clouds), Emerald.
M:N	Combina atributos de M:1 y casos 1:M.	TRIX.

Relación muchos-a-muchos. La idea de tener una relación muchos-a-muchos entre hilos y procesos ha sido explorada en el sistema operativo experimental TRIX [SIEB83,WARD80]. En TRIX, existen los conceptos de dominio de hilo. Un dominio es una entidad estática, que consiste en un espacio de direcciones y «puertos» a través de los cuales se pueden enviar y recibir mensajes. Un hilo es una ruta de ejecución, con una pila de ejecución, estado del procesador e información de planificación.

Al igual que en el enfoque multihilo visto hasta el momento, múltiples hilos podrían ejecutar en un solo dominio, proporcionando las ventajas discutidas anteriormente. Sin embargo, también es posible realizar la actividad de usuario o ejecutar aplicaciones en múltiples dominios. En este caso hay un hilo que se puede mover entre dominios.

El uso de un solo hilo en múltiples dominios está motivado por el deseo de proporcionar herramientas de estructuración al programador. Por ejemplo, considere un programa que hace uso de un

subprograma de E/S. En un entorno multiprogramado que permite procesos creados por los usuarios, el programa principal podría generar un nuevo proceso para el manejo de la E/S y continuar ejecutando. Sin embargo, si el futuro progreso del programa principal depende del funcionamiento de la E/S, entonces el programa principal tendrá que esperar a la finalización del otro programa de E/S. Hay varias formas de implementar esta aplicación:

1. El programa completo puede implementarse como un solo proceso. Esta solución es razonable y directa. Sin embargo, existen desventajas relativas a la gestión de memoria. El proceso completo podría requerir una gran cantidad de memoria para ejecutar de forma eficiente, mientras que el subprograma de E/S requiere relativamente poco espacio de direccionamiento para almacenar la E/S y para manejar su poca cantidad de código fuente. Ya que el programa de E/S ejecuta en el espacio de direcciones del programa mayor, puede suceder que el proceso completo deba permanecer en la memoria principal durante el funcionamiento de la E/S o que la operación de E/S esté sujeta a intercambio. Este efecto sobre la gestión de memoria también existiría si el programa principal y el subprograma hubieran sido implementados como dos hilos en el mismo espacio de direcciones.

2. El programa principal y el subprograma de E/S podrían implementarse como dos procesos independientes. Este enfoque presenta la sobrecarga de la creación del proceso subordinado. Si la actividad de E/S es frecuente, se deberá dejar vivo al proceso subordinado, con el consiguiente consumo de recursos, o se deberá crear y destruir frecuentemente el subprograma, con la consiguiente pérdida de eficiencia.

3. Tratar al programa principal y al subprograma de E/S como una sola actividad que se puede implementar en un solo hilo. Sin embargo, se podría crear un espacio de direcciones (dominio) para el programa principal y otro para el subprograma de E/S. De esta forma, el hilo se podría mover entre los dos espacios de direcciones a lo largo de la ejecución. El sistema operativo puede gestionar los dos espacios de direcciones de forma independiente, y no se genera una sobrecarga de creación de procesos. Adicionalmente, el espacio de direcciones utilizado por el subprograma de E/S también podría ser compartido por otros programas de E/S.

Las experiencias de los desarrolladores de TRIX indican que la tercera opción tiene mérito y que podría ser la más eficiente para algunas aplicaciones.

Relación uno-a-muchos. En el campo de los sistemas operativos distribuidos (diseñados para controlar sistemas de computadores distribuidos), ha habido interés en el concepto de un hilo como una entidad que se puede mover entre espacios de direcciones[5]. Un ejemplo importante de esta investigación es el sistema operativo Clouds, y especialmente su núcleo, conocido como Ra [DASG92]. Otro ejemplo es el sistema Emerald [STEE95].

Desde el punto de vista del usuario, un hilo en Clouds es una unidad de actividad. Un proceso es un espacio de direcciones virtual con un bloque de control de proceso asociado. Una vez creado, un hilo comienza ejecutando en un proceso a través de la invocación de un punto de entrada de un programa en ese proceso. Los hilos se pueden mover de un espacio de direcciones a otro, incluso fuera de los límites de la máquina (es decir, moverse de un computador a otro). Según se mueve el hilo, debe llevarse determinada información con él, tal como el controlador de terminal, los parámetros globales y las guías de planificación (por ejemplo, prioridad).

---

[5] El movimiento de procesos o hilos entre espacios de direcciones, o migración de hilos entre máquinas diferentes, se ha convertido en un tema de interés en los últimos años. Este tema se analiza en el Capítulo 14.

El enfoque de Clouds proporciona una forma eficiente de aislar a los usuarios y programadores de los detalles del entorno distribuido. La actividad del usuario puede ser representada como un solo hilo, y el movimiento de ese hilo entre máquinas puede ser gestionado por el sistema operativo gracias a información relativa al sistema, tal como la necesidad de acceder a un recurso remoto o el equilibrado de carga.

4.2. MULTIPROCESAMIENTO SIMÉTRICO

Tradicionalmente, el computador ha sido visto como una máquina secuencial. La mayor parte de los lenguajes de programación requieren que el programador especifique algoritmos como una secuencia de instrucciones. Un procesador ejecuta programas a través de la ejecución de instrucciones máquina en secuencia y de una en una. Cada instrucción se ejecuta como una secuencia de operaciones (ir a buscar la instrucción, ir a buscar los operandos, realizar la operación, almacenar resultados).

Esta visión del computador nunca ha sido totalmente cierta. A nivel de micro-operación, se generan múltiples señales de control al mismo tiempo. El *pipeline* de instrucciones, al menos en lo relativo a la búsqueda y ejecución de operaciones, ha estado presente durante mucho tiempo. Éstos son dos ejemplos de realización de funciones en paralelo.

A medida que ha evolucionado la tecnología de los computadores y el coste del hardware ha descendido, los diseñadores han visto cada vez más oportunidades para el paralelismo, normalmente para mejorar el rendimiento y, en algunos casos, para mejorar la fiabilidad. En este libro, examinamos los dos enfoques más populares para proporcionar paralelismo a través de la réplica de procesadores: multiprocesamiento simétricos (SMP) y *clusters*. Los SMP se abordan en esta sección; los *clusters* se examinan en la Parte Seis.

ARQUITECTURA SMP

Es útil ver donde encaja la arquitectura SMP dentro de las categorías de procesamiento paralelo. La forma más común de categorizar estos sistemas es la taxonomía de sistemas de procesamiento paralelo introducida por Flynn [FLYN72]. Flynn propone las siguientes categorías de sistemas de computadores:

- **Única instrucción, único flujo de datos – *Single instruction single data* (SISD) *stream*.** Un solo procesador ejecuta una única instrucción que opera sobre datos almacenados en una sola memoria.
- **Única instrucción, múltiples flujos de datos – *Single instruction multiple data* (SIMD) *stream*.** Una única instrucción de máquina controla la ejecución simultánea de un número de elementos de proceso. Cada elemento de proceso tiene una memoria de datos asociada, de forma que cada instrucción se ejecuta en un conjunto de datos diferente a través de los diferentes procesadores. Los procesadores vectoriales y matriciales entran dentro de esta categoría.
- **Múltiples instrucciones, único flujo de datos – *Multiple instruction single data* (MISD) *stream*.** Se transmite una secuencia de datos a un conjunto de procesadores, cada uno de los cuales ejecuta una secuencia de instrucciones diferente. Esta estructura nunca se ha implementado.
- **Múltiples instrucciones, múltiples flujos de datos – *Multiple instruction multiple data* (MIMD) *stream*.** Un conjunto de procesadores ejecuta simultáneamente diferentes secuencias de instrucciones en diferentes conjuntos de datos.

Con la organización MIMD, los procesadores son de propósito general, porque deben ser capaces de procesar todas las instrucciones necesarias para realizar las transformaciones de datos apropiadas. MIMD se puede subdividir por la forma en que se comunican los procesadores (Figura 4.8). Si cada procesador tiene una memoria dedicada, cada elemento de proceso es en sí un computador. La comunicación entre los computadores se puede realizar a través de rutas prefijadas o bien a través de redes. Este sistema es conocido como un ***cluster***, o multicomputador. Si los procesadores comparten una memoria común, entonces cada procesador accede a los programas y datos almacenados en la memoria compartida, y los procesadores se comunican entre sí a través de dicha memoria; este sistema se conoce como **multiprocesador de memoria compartida**.

Una clasificación general de los multiprocesadores de memoria compartida se basa en la forma de asignar procesos a los procesadores. Los dos enfoques fundamentales son maestro/esclavo y simétrico. Con la arquitectura **maestro/esclavo**, el núcleo del sistema operativo siempre ejecuta en un determinado procesador. El resto de los procesadores sólo podrán ejecutar programas de usuario y, a lo mejor, utilidades del sistema operativo. El maestro es responsable de la planificación de procesos e hilos. Una vez que un proceso/hilo está activado, si el esclavo necesita servicios (por ejemplo, una llamada de E/S), debe enviar una petición al maestro y esperar a que se realice el servicio. Este enfoque es bastante sencillo y requiere pocas mejoras respecto a un sistema operativo multiprogramado uniprocesador. La resolución de conflictos se simplifica porque un procesador tiene el control de toda la memoria y recursos de E/S. Las desventajas de este enfoque son las siguientes:

- Un fallo en el maestro echa abajo todo el sistema.
- El maestro puede convertirse en un cuello de botella desde el punto de vista del rendimiento, ya que es el único responsable de hacer toda la planificación y gestión de procesos.

En un **multiprocesador simétrico (*Symmetric Multiprocessor*, SMP)**, el núcleo puede ejecutar en cualquier procesador, y normalmente cada procesador realiza su propia planificación del conjunto disponible de procesos e hilos. El núcleo puede construirse como múltiples procesos o múltiples hilos, permitiéndose la ejecución de partes del núcleo en paralelo. El enfoque SMP complica al sistema operativo, ya que debe asegurar que dos procesadores no seleccionan un mismo proceso y que no se pierde ningún proceso de la cola. Se deben emplear técnicas para resolver y sincronizar el uso de los recursos.

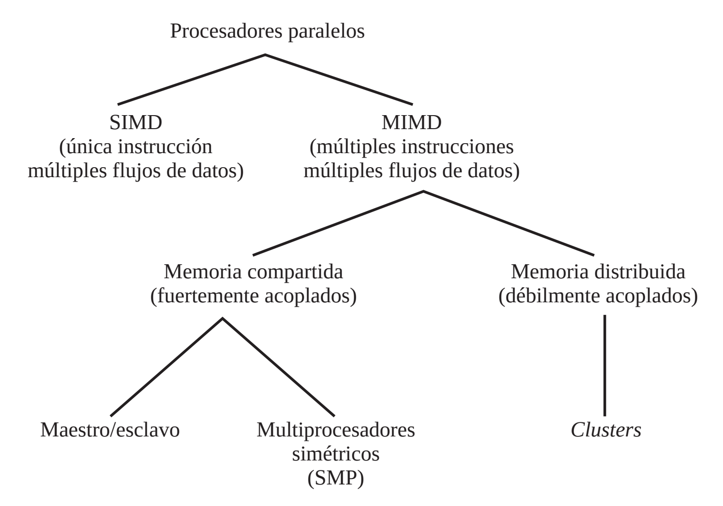


Figura 4.8. Arquitectura de procesadores paralelos.

El diseño de SMP y *clusters* es complejo, e involucra temas relativos a la organización física, estructuras de interconexión, comunicación entre procesadores, diseño del sistema operativo y técnicas de aplicaciones software. Nuestra preocupación aquí, y más adelante cuando hablemos de los clústeres (Capítulo 13), se centra en los aspectos de diseño del sistema operativo, aunque en ambos casos veremos brevemente temas de organización.

ORGANIZACIÓN SMP

La Figura 4.9 muestra la organización general de un SMP. Existen múltiples procesadores, cada uno de los cuales contiene su propia unidad de control, unidad aritmético-lógica y registros. Cada procesador tiene acceso a una memoria principal compartida y dispositivos de E/S a través de algún mecanismo de interconexión; el bus compartido es común a todos los procesadores. Los procesadores se pueden comunicar entre sí a través de la memoria (mensajes e información de estado dejados en espacios de memoria compartidos). Los procesadores han de poder intercambiarse señales directamente. A menudo la memoria está organizada de tal manera que se pueden realizar múltiples accesos simultáneos a bloques separados.

En máquinas modernas, los procesadores suelen tener al menos un nivel de memoria cache, que es privada para el procesador. El uso de esta cache introduce nuevas consideraciones de diseño. Debido a que la cache local contiene la imagen de una porción de memoria principal, si se altera una palabra en una cache, se podría invalidar una palabra en el resto de las caches. Para prevenir esto, el resto de los procesadores deben ser alertados de que se ha llevado a cabo una actualización. Este problema se conoce como el problema de coherencia de caches y se suele solucionar con técnicas hardware más que con el sistema operativo[6].

CONSIDERACIONES DE DISEÑO DE SISTEMAS OPERATIVOS MULTIPROCESADOR

Un sistema operativo SMP gestiona los procesadores y otros recursos del computador, de manera que el usuario puede ver al sistema de la misma forma que si fuera un sistema uniprocesador multiprogramado. Un usuario puede desarrollar aplicaciones que utilicen múltiples procesos o múltiples hilos dentro de procesos sin preocuparse de si estará disponible un único procesador o múltiples procesadores. De esta forma, un sistema operativo multiprocesador debe proporcionar toda la funcionalidad de un sistema multiprogramado, además de características adicionales para adecuarse a múltiples procesadores. Las principales claves de diseño incluyen las siguientes características:

- **Procesos o hilos simultáneos concurrentes.** Las rutinas del núcleo necesitan ser reentrantes para permitir que varios procesadores ejecuten el mismo código del núcleo simultáneamente. Debido a que múltiples procesadores pueden ejecutar la misma o diferentes partes del código del núcleo, las tablas y la gestión de las estructuras del núcleo deben ser gestionas apropiadamente para impedir interbloqueos u operaciones inválidas.
- **Planificación.** La planificación se puede realizar por cualquier procesador, por lo que se deben evitar los conflictos. Si se utiliza multihilo a nivel de núcleo, existe la posibilidad de planificar múltiples hilos del mismo proceso simultáneamente en múltiples procesadores. En el Capítulo 10 se examina la planificación multiprocesador.

---

[6] En [STAL03] se proporciona una descripción de esquemas de coherencia de cache basados en el hardware.

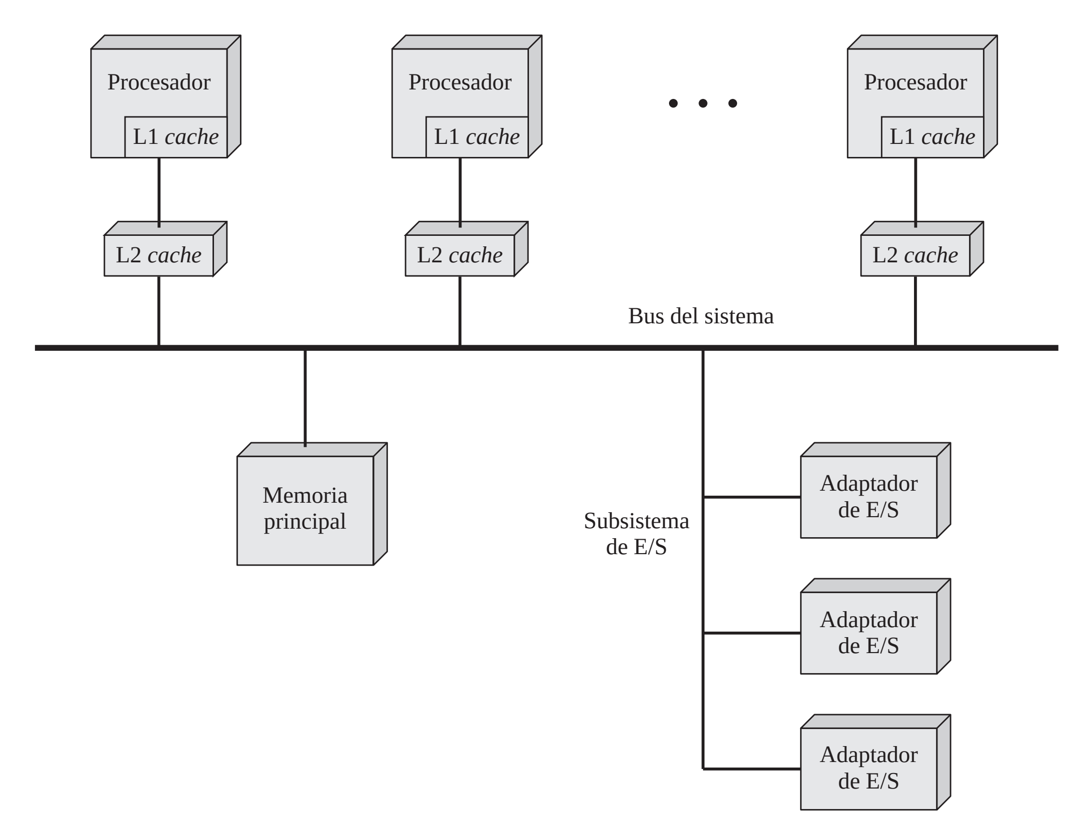


Figura 4.9. Organización de los multiprocesadores simétricos.

- **Sincronización.** Con múltiples procesos activos, que pueden acceder a espacios de direcciones compartidas o recursos compartidos de E/S, se debe tener cuidado en proporcionar una sincronización eficaz. La sincronización es un servicio que fuerza la exclusión mutua y el orden de los eventos. Un mecanismo común de sincronización que se utiliza en los sistemas operativos multiprocesador son los cerrojos, descritos en el Capítulo 5.
- **Gestión de memoria.** La gestión de memoria en un multiprocesador debe tratar con todos los aspectos encontrados en las máquinas uniprocesador, que se verán en la Parte Tres. Además, el sistema operativo necesita explotar el paralelismo hardware existente, como las memorias multipuerto, para lograr el mejor rendimiento. Los mecanismos de paginación de los diferentes procesadores deben estar coordinados para asegurar la consistencia cuando varios procesadores comparten una página o segmento y para decidir sobre el reemplazo de una página.
- **Fiabilidad y tolerancia a fallos.** El sistema operativo no se debe degradar en caso de fallo de un procesador. El planificador y otras partes del sistema operativo deben darse cuenta de la pérdida de un procesador y reestructurar las tablas de gestión apropiadamente.

Debido a que los aspectos de diseño de un sistema operativo multiprocesador suelen ser extensiones a soluciones de problemas de diseño de uniprocesadores multiprogramados, no los trataremos por separado. En su lugar, los aspectos específicos a los temas multiprocesador se tratarán en su contexto apropiado a lo largo del libro.

4.3. MICRONÚCLEOS

Un concepto que últimamente está recibiendo mucha atención es el de micronúcleo. Un micronúcleo es la pequeña parte central de un sistema operativo que proporciona las bases para extensiones modulares. Sin embargo, el término es algo confuso, y hay varias cuestiones relacionadas con los micronúcleos con respuestas distintas por parte de diferentes equipos de diseño de sistemas operativos. Estas cuestiones incluyen, cómo de pequeño debe ser un núcleo para denominarse micronúcleo, cómo diseñar manejadores de dispositivos para obtener el mejor rendimiento a la vez que se abstraen sus funciones del hardware, si ejecutar operaciones que no pertenecen al núcleo dentro de éste o en el espacio de usuario, y si mantener el código de subsistemas existentes (por ejemplo, una versión de UNIX) o empezar de cero.

El enfoque de micronúcleo se popularizó por su uso en el sistema operativo Mach. En teoría este enfoque proporciona un alto grado de flexibilidad y modularidad. Determinados productos ya tienen implementaciones micronúcleo, y este enfoque general de diseño se verá en la mayor parte de los computadores personales, estaciones de trabajo, y sistemas operativos servidor que se desarrollen en un futuro cercano.

ARQUITECTURA MICRONÚCLEO

Los primeros sistemas operativos desarrollados a mediados y finales de los años 50 fueron diseñados sin preocuparse por su arquitectura. Nadie tenía la experiencia necesaria en construcción de sistemas software realmente grandes, y los problemas causados por la dependencia mutua e interacción no se tenían en cuenta. En estos **sistemas operativos monolíticos**, de prácticamente cualquier procedimiento se podía llamar a cualquier otro. Esta falta de estructura se hizo insostenible a medida que los sistemas operativos crecieron hasta proporciones desmesuradas. Por ejemplo, la primera versión de OS/360 contenía más de un millón de líneas del código; Multics, desarrollado más tarde, creció hasta 20 millones de líneas del código [DENN84]. Como vimos en la Sección 2.3, se necesitaron técnicas de programación modular para manejar esta escala de desarrollo software. Específicamente, se desarrollaron los **sistemas operativos por capas**[7] (Figura 4.10a), en los cuales las funciones se organizan jerárquicamente y sólo hay interacción entre las capas adyacentes. Con el enfoque por capas, la mayor parte o todas las capas ejecutan en modo núcleo.

Los problemas permanecen incluso en el enfoque por capas. Cada capa posee demasiada funcionalidad y grandes cambios en una capa pueden tener numerosos efectos, muchos difíciles de seguir, en el código de las capas adyacentes (encima o debajo). Como resultado es difícil implementar versiones a medida del sistema operativo básico con algunas funciones añadidas o eliminadas. Además, es difícil construir la seguridad porque hay muchas interacciones entre capas adyacentes.

La filosofía existente en el micronúcleo es que solamente las funciones absolutamente esenciales del sistema operativo estén en el núcleo. Los servicios y aplicaciones menos esenciales se construyen sobre el micronúcleo y se ejecutan en modo usuario. Aunque la filosofía de qué hay dentro y qué hay fuera del micronúcleo varía de un diseño a otro, la característica general es que muchos servicios que tradicionalmente habían formado parte del sistema operativo ahora son subsistemas ex-

---

[7] Como de costumbre, la terminología en esta área no se aplica de forma consistente en la literatura. El término *sistema operativo monolítico* se utiliza frecuentemente para referirse a los dos tipos de sistemas operativos que hemos denominado como *monolíticos* y *por capas*.

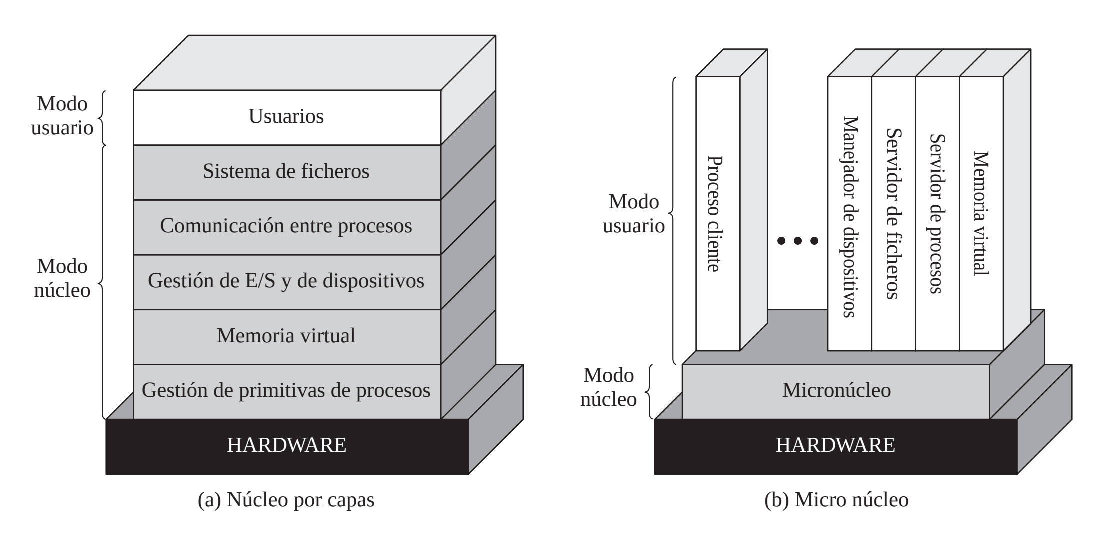


Figura 4.10. Arquitectura del núcleo.

ternos que interactúan con el núcleo y entre ellos mismos; algunos ejemplos son: manejadores de dispositivos, servidores de archivos, gestores de memoria virtual, sistemas de ventana y servicios de seguridad.

La arquitectura del micronúcleo reemplaza la tradicional estructura vertical y estratificada en capas por una horizontal (Figura 4.10b). Los componentes del sistema operativo externos al micronúcleo se implementan como servidores de procesos; interactúan entre ellos dos a dos, normalmente por paso de mensajes a través del micronúcleo. De esta forma, el micronúcleo funciona como un intercambiador de mensajes: válida mensajes, los pasa entre los componentes, y concede el acceso al hardware. El micronúcleo también realiza una función de protección; previene el paso de mensajes a no ser que el intercambio esté permitido.

Por ejemplo, si una aplicación quiere abrir un archivo, manda un mensaje al servidor del sistema de archivos. Si quiere crear un proceso o hilo, manda un mensaje al servidor de procesos. Cada uno de los servidores puede mandar mensajes al resto de los servidores y puede invocar funciones primitivas del micronúcleo. Es decir, es un arquitectura cliente/servidor dentro de un solo computador.

BENEFICIOS DE UNA ORGANIZACIÓN MICRONÚCLEO

En la literatura se pueden encontrar una serie de ventajas del uso de los micronúcleos (por ejemplo, [FINK97], [LIED96a], [WAYN94a]. Estas incluyen:

- Interfaces uniformes
- Extensibilidad
- Flexibilidad
- Portabilidad
- Fiabilidad
- Soporte de sistemas distribuidos
- Soporte de sistemas operativos orientados a objetos (OOOS)

El micronúcleo impone una **interfaz uniforme** en las peticiones realizadas por un proceso. Los procesos no necesitan diferenciar entre servicios a nivel de núcleo y a nivel de usuario, porque todos los servicios se proporcionan a través de paso de mensajes.

De forma inevitable, un sistema operativo tendrá que adquirir nuevas características que no están en su diseño actual, a medida que se desarrollen nuevos dispositivos hardware y nuevas técnicas software. La arquitectura micronúcleo facilita la **extensibilidad**, permitiendo agregar nuevos servicios, así como la realización de múltiples servicios en la misma área funcional. Por ejemplo, puede haber múltiples organizaciones de archivos para disquetes; cada organización se puede implementar como un proceso a nivel de usuario más que tener múltiples servicios de archivos disponibles en el núcleo. De esta forma, los usuarios pueden elegir, de una variedad de servicios, el que mejor se adapte a sus necesidades. Con la arquitectura micronúcleo, cuando se añade una nueva característica, sólo el servidor relacionado necesita modificarse o añadirse. El impacto de un servidor nuevo o modificado se restringe a un subconjunto del sistema. Además, las modificaciones no requieren la construcción de un nuevo núcleo.

Relacionado con la extensibilidad de una arquitectura micronúcleo está su **flexibilidad**. No sólo se pueden añadir nuevas características al sistema operativo, además las características existentes se pueden eliminar para realizar una implementación más pequeña y más eficiente. Un sistema operativo micronúcleo no es necesariamente un sistema pequeño. De hecho, su propia estructura le permite añadir un amplio rango de características. Pero no todo el mundo necesita, por ejemplo, un alto nivel de seguridad o la necesidad de realizar computación distribuida. Si las características sustanciales (en términos de requisitos de memoria) se hacen opcionales, el producto será atractivo para mayor número de usuarios.

El monopolio que casi tiene Intel en muchos segmentos del mercado de la computación es improbable que dure indefinidamente. De esta forma, la **portabilidad** se convierte en una característica interesante en los sistemas operativos. En la arquitectura micronúcleo, todo o gran parte del código específico del procesador está en el micronúcleo. Por tanto, los cambios necesarios para transferir el sistema a un nuevo procesador son menores y tienden a estar unidos en grupos lógicos.

Mayor sea el tamaño de un producto software, mayor es la dificultad de asegurar su fiabilidad. Aunque el diseño modular ayuda a mejorar la **fiabilidad**, con una arquitectura micronúcleo se pueden lograr mayores ganancias. Un micronúcleo pequeño se puede verificar de forma rigurosa. El que sólo utilice un pequeño número de interfaces de programación de aplicaciones (API) hace más sencillo producir un código de calidad para los servicios del sistema operativo fuera del núcleo. El programador del sistema tiene un número limitado de API que dominar y métodos limitados de interacción y, por consiguiente, es más difícil afectar negativamente a otros componentes del sistema.

El micronúcleo nos lleva por sí mismo al **soporte de sistemas distribuidos**, incluyendo *clusters* controlados por sistemas operativos distribuidos. Cuando se envía un mensaje desde un cliente hasta un proceso servidor, el mensaje debe incluir un identificador del servicio pedido. Si se configura un sistema distribuido (por ejemplo, un *cluster*) de tal forma que todos los procesos y servicios tengan identificadores únicos, entonces habrá una sola imagen del sistema a nivel de micronúcleo. Un proceso puede enviar un mensaje sin saber en qué máquina reside el servicio pedido. Volveremos a este punto en nuestra discusión de los sistemas distribuidos en la Parte Seis.

Una arquitectura micronúcleo funciona bien en el contexto de un **sistema operativo orientado a objetos**. Un enfoque orientado a objetos puede servir para diseñar el micronúcleo y para desarrollar extensiones modulares para el sistema operativo. Como resultado, varios diseños de micronúcleos van en la dirección de orientación a objetos [WAYN94b]. Un enfoque prometedor para casar la arquitectura micronúcleo con los principios de OOOS es el uso de componentes [MESS96]. Los componentes son objetos con interfaces claramente definidas que pueden ser interconectadas para la realiza-

ción de software a través de bloques de construcción. Toda la interacción entre componentes utiliza la interfaz del componente. Otros sistemas, tales como Windows, no se basan exclusivamente o por completo en métodos orientados a objetos pero han incorporado los principios de la orientación a objetos en el diseño del micronúcleo.

RENDIMIENTO DEL MICRONÚCLEO

Una potencial desventaja que se cita a menudo de los micronúcleos es la del rendimiento. Lleva más tiempo construir y enviar un mensaje a través del micronúcleo, y aceptar y decodificar la respuesta, que hacer una simple llamada a un servicio. Sin embargo, también son importantes otros factores, de forma que es difícil generalizar sobre la desventaja del rendimiento, si es que la hay.

Hay mucho que depende del tamaño y de la funcionalidad del micronúcleo. [LIED96a] resume un número de estudios que revelan una pérdida sustancial del rendimiento en los que pueden ser denominados micronúcleos de primera generación. Estas pérdidas continúan a pesar de los esfuerzos realizados para optimizar el código del micronúcleo. Una respuesta a este problema fue hacer mayores los micronúcleos, volviendo a introducir servicios críticos y manejadores en el sistema operativo. Los primeros ejemplos de este enfoque son Mach y Chorus. Incrementando de forma selectiva la funcionalidad del micronúcleo se reduce el número de cambios de modo usuario-núcleo y el número de cambios de espacio de direcciones de proceso. Sin embargo, esta solución reduce los problemas de rendimiento sacrificando la fortaleza del diseño del micronúcleo: mínimas interfaces, flexibilidad, etc.

Otro enfoque consiste en hacer el micronúcleo, no más grande, sino más pequeño. [LIED96b] argumenta que, apropiadamente diseñado, un micronúcleo muy pequeño elimina las pérdidas de rendimiento y mejora la flexibilidad y fiabilidad. Para dar una idea de estos tamaños, el típico micronúcleo de primera generación tenía 300 Kbytes de código y 140 interfaces de llamadas al sistema. Un ejemplo de un micronúcleo pequeño de segunda generación es el L4 [HART97, LIED95], que consiste en 12 Kbytes de código y 7 llamadas al sistema. Las experimentaciones realizadas en estos sistemas indican que pueden funcionar tan bien o mejor que sistemas operativos por capas como por ejemplo, UNIX.

DISEÑO DEL MICRONÚCLEO

Debido a que los diferentes micronúcleos presentan una gran variedad de tamaños y funcionalidades, no se pueden facilitar reglas concernientes a qué funcionalidades deben darse por parte del micronúcleo y qué estructura debe implementarse. En esta sección, presentamos un conjunto mínimo de funciones y servicios del micronúcleo, para dar una perspectiva del diseño de los mismos.

El micronúcleo debe incluir aquellas funciones que dependen directamente del hardware y aquellas funciones necesarias para mantener a los servidores y aplicaciones operando en modo usuario. Estas funciones entran dentro de las categorías generales de gestión de memoria a bajo nivel, intercomunicación de procesos (IPC), y E/S y manejo de interrupciones.

Gestión de memoria a bajo nivel. El micronúcleo tiene que controlar el concepto hardware de espacio de direcciones para hacer posible la implementación de protección a nivel de proceso. Con tal de que el micronúcleo se responsabilice de la asignación de cada página virtual a un marco físico, la parte principal de gestión de memoria, incluyendo la protección del espacio de memoria entre procesos, el algoritmo de reemplazo de páginas y otra lógica de paginación, pueden implementarse fuera del núcleo. Por ejemplo, un módulo de memoria virtual fuera del micronúcleo decide cuándo traer

una página a memoria y qué página presente en memoria debe reemplazarse; el micronúcleo proyecta estas referencias de página en direcciones físicas de memoria principal.

El paginador externo de Mach [YOUN87] introdujo el concepto de que la paginación y la gestión de la memoria virtual se pueden realizar de forma externa al núcleo. La Figura 4.11 muestra el funcionamiento del paginador externo. Cuando un hilo de la aplicación hace referencia a una página que no está en memoria principal, hay un fallo de página y la ejecución pasa al núcleo. El núcleo entonces manda un mensaje al proceso paginador indicando la página que ha sido referenciada. El paginador puede decidir cargar esa página y localizar un marco de página para este propósito. El paginador y el núcleo deben interactuar para realizar las operaciones lógicas del paginador en memoria física. Una vez que la página está disponible, el paginador manda un mensaje a la aplicación a través del micronúcleo.

Esta técnica permite a un proceso fuera del núcleo proyectar archivos y bases de datos en espacios de direcciones de usuario sin llamar al núcleo. Las políticas de compartición de memoria específicas de la aplicación se pueden implementar fuera del núcleo.

[LIED95] recomienda un conjunto de tres operaciones de micronúcleo que pueden dar soporte a la paginación externa y a la gestión de memoria virtual:

- **Conceder (*Grant*).** El propietario de un espacio de direcciones (un proceso) puede conceder alguna de sus páginas a otro proceso. El núcleo borra estas páginas del espacio de memoria del otorgante y se las asigna al proceso especificado.
- **Proyectar (*Map*).** Un proceso puede proyectar cualquiera de sus páginas en el espacio de direcciones de otro proceso, de forma que ambos procesos tienen acceso a las páginas. Esto genera memoria compartida entre dos procesos. El núcleo mantiene la asignación de estas páginas al propietario inicial, pero proporciona una asociación que permite el acceso de otros procesos.
- **Limpiar (*Flush*).** Un proceso puede reclamar cualquier página que fue concedida o asociada a otro proceso.

Para comenzar, el núcleo asigna toda la memoria física disponible como recursos a un proceso base del sistema. A medida que se crean los nuevos procesos, se pueden conceder o asociar algunas páginas del espacio de direcciones original a los nuevos procesos. Este esquema podría dar soporte a múltiples esquemas de memoria virtual simultáneamente.

Comunicación entre procesos (*Interprocess Communication*). La forma básica de comunicación entre dos procesos o hilos en un sistema operativo con micronúcleo son los **mensajes**. Un

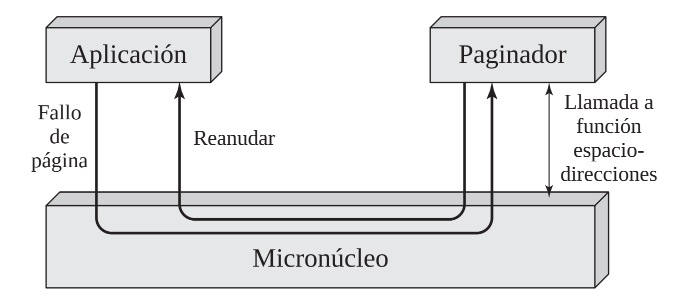


Figura 4.11. Procesamiento del fallo de página.

mensaje incluye una cabecera que identifica a los procesos remitente y receptor y un cuerpo que contiene directamente los datos, un puntero a un bloque de datos, o alguna información de control del proceso. Normalmente podemos pensar que las IPC se fundamentan en puertos asociados a cada proceso. Un **puerto** es, en esencia, una cola de mensajes destinada a un proceso particular; un proceso puede tener múltiples puertos. Asociada a cada puerto existe una lista que indica qué procesos se pueden comunicar con éste. Las identidades y funcionalidades de cada puerto se mantienen en el núcleo. Un proceso puede concederse nuevas funcionalidades mandando un mensaje al núcleo con las nuevas funcionalidades del puerto.

Llegado a este punto sería conveniente realizar un comentario sobre el paso de mensajes. El paso de mensajes entre dos procesos que no tengan solapado el espacio de direcciones implica realizar una copia de memoria a memoria, y de esta forma está limitado por la velocidad de la memoria y no se incrementa con la velocidad del procesador. De esta forma, la investigación actual de los sistemas operativos muestra interés en IPC basado en hilos y esquemas de memoria compartida como la re-proyección de páginas —*page remapping*— (una sola página compartida por múltiples procesos).

Gestión de E/S e interrupciones. Con una arquitectura micronúcleo es posible manejar las interrupciones hardware como mensajes e incluir los puertos de E/S en los espacios de direcciones. El micronúcleo puede reconocer las interrupciones pero no las puede manejar. Más bien, genera un mensaje para el proceso a nivel de usuario que está actualmente asociado con esa interrupción. De esta forma, cuando se habilita una interrupción, se asigna un proceso de nivel de usuario a esa interrupción y el núcleo mantiene las asociaciones. La transformación de las interrupciones en mensajes las debe realizar el micronúcleo, pero el micronúcleo no está relacionado con el manejo de interrupciones específico de los dispositivos.

[LIED96a] sugiere ver el hardware como un conjunto de hilos que tienen identificadores de hilo único y que mandan mensajes (únicamente con el identificador de hilo) a hilos asociados en el espacio de usuario. El hilo receptor determina si el mensaje proviene de una interrupción y determina la interrupción específica. La estructura general de este código a nivel de usuario es la siguiente:

```
hilo del dispositivo:
   do
          waitFor(msg, remitente);
          if (remitente == mi_interrupcion_hardware)
          {
                 leer/escribir puertos E/S;
                 reanudar interrupción hardware;
          }
          else ...
   while(true);
```

4.4. GESTIÓN DE HILOS Y SMP EN WINDOWS

El diseño de un proceso Windows está limitado por la necesidad de proporcionar soporte a diversos entornos de sistemas operativos. Los procesos soportados por diferentes entornos de sistemas operativos se diferencian en varias cosas, incluyendo las siguientes:

- La denominación de los procesos.
- Si se proporcionan hilos con los procesos.
- Cómo se representa a los procesos.
- Cómo se protege a los recursos de los procesos.
- Qué mecanismos se utilizan para la comunicación y sincronización entre procesos.
- Cómo se relacionan los procesos entre sí.

Como consecuencia, las estructuras de los procesos y los servicios proporcionados por el núcleo de Windows son relativamente sencillos y de propósito general, permitiendo a cada subsistema del sistema operativo que emule una estructura y funcionalidad particular del proceso. Algunas características importantes de los procesos Windows son las siguientes:

- Los procesos Windows están implementados como objetos.
- Un proceso ejecutable puede contener uno o más hilos.
- Tanto el objeto proceso como el objeto hilo, tienen funcionalidades de sincronización preconstruidas.

La Figura 4.12, basada en una de [SOLO00], muestra la forma en la que un proceso se asocia a los recursos que controla o utiliza. A cada proceso se le asigna un testigo (*token*) de seguridad de acceso, denominada la ficha principal del proceso. Cuando un usuario inicia una sesión, Windows crea una ficha de acceso que incluye el ID de seguridad para el usuario. Cada proceso que se crea o ejecuta en representación de este usuario, tiene una copia de este testigo de acceso. Windows lo utiliza para comprobar si el usuario puede acceder a objetos de seguridad, o puede realizar funciones restringidas en el sistema o en un objeto de seguridad. El testigo de acceso controla si un proceso puede modificar sus propios atributos. En este caso, el proceso no tiene un manejador abierto hacia su testigo de acceso. Si el proceso intenta abrir este manejador, el sistema de seguridad determinará si está permitido y por tanto si el proceso puede modificar sus propios atributos.

También relacionado con el proceso, hay una serie de bloques que definen el espacio de direcciones virtuales actualmente asignado al proceso. El proceso no puede modificar directamente estas estructuras, ya que dependen del gestor de memoria virtual, que proporciona servicios de asignación de memoria a los procesos.

Finalmente, el proceso incluye una tabla de objetos, que trata los otros objetos conocidos por el proceso. Hay un manejador para cada hilo que está contenido en este objeto. La Figura 4.12 muestra un único hilo. El proceso tiene acceso a un objeto archivo y a un objeto segmento, que define un segmento de memoria compartido.

OBJETO PROCESO Y OBJETO HILO

La estructura orientada a objetos de Windows facilita el desarrollo de un proceso de propósito general. Windows hace uso de dos tipos de objetos relacionados con los procesos: procesos e hilos. Un proceso es una entidad que corresponde a un trabajo de usuario o una aplicación que posee recursos como la memoria y archivos abiertos. Un hilo es una unidad de trabajo que se puede activar, que ejecuta secuencialmente y que es interrumpible, de forma que el procesador puede cambiar a otro hilo.

Cada proceso Windows se representa por un objeto. En la Figura 4.23a se puede ver la estructura de dicho objeto. Un proceso se define por una serie de atributos y encapsula una serie de acciones, o servicios, que puede realizar. Un proceso realizará un servicio cuando reciba el mensaje apropiado; la

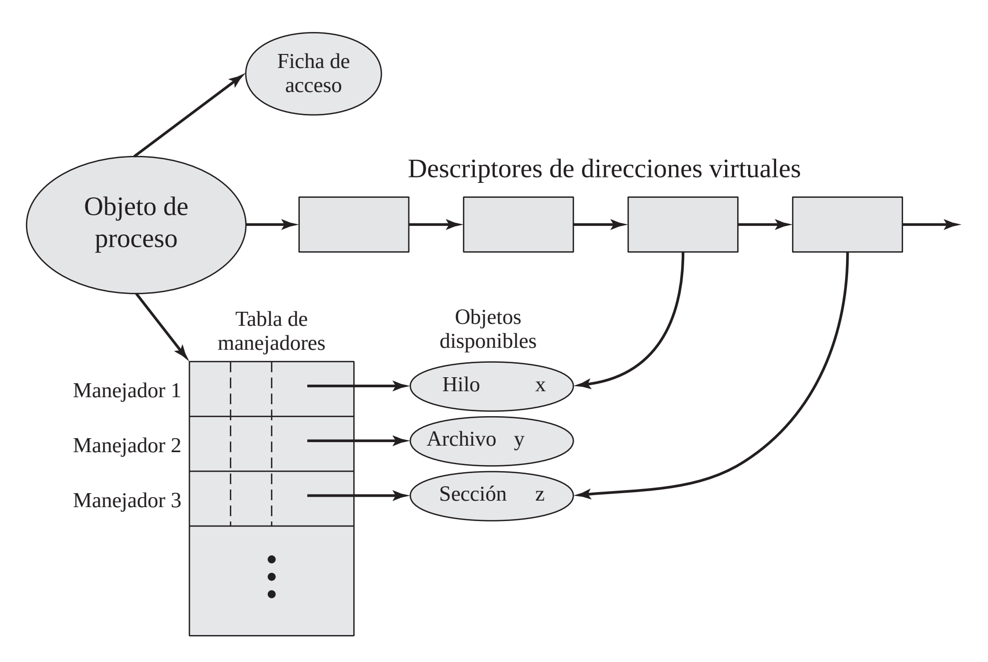


Figura 4.12. Un proceso Windows y sus recursos.

única forma de invocar a este servicio es a través de mensajes a un objeto proceso que proporciona ese servicio. Cuando Windows crea un nuevo proceso, utiliza el objeto definido para el proceso Windows como plantilla para generar la nueva instancia del proceso. En el momento de la creación se asignan los valores de los atributos. La Tabla 4.3 da una breve definición de cada atributo de objeto contenido en el objeto proceso.

Un proceso Windows debe contener por lo menos un hilo que ejecutar. Ese hilo puede a su vez crear otros hilos. En un sistema multiprocesador, múltiples hilos de un mismo proceso pueden ejecutar en paralelo. La Figura 4.13b representa la estructura de objeto hilo, y la Tabla 4.4 define los atributos del objeto hilo. Es importante darse cuenta de que alguno de los atributos de los hilos se asemeja a los de los procesos. En estos casos, el valor del atributo del hilo se deriva del valor del atributo del proceso. Por ejemplo, la *afinidad de procesador asociada al hilo* (*thread processor affinity*) es el conjunto de procesadores, en un sistema multiprocesador, que pueden ejecutar este hilo. Este conjunto es igual o un subconjunto de la *afinidad de procesador asociada al proceso*.

Adviértase que uno de los atributos del proceso hilo es el *contexto*. Esta información permite que el hilo se suspenda y reanude. Más aún, es posible cambiar el comportamiento de un hilo, alterando su contenido cuando el hilo está suspendido.

MULTIHILO

Windows soporta la concurrencia entre procesos ya que hilos de diferentes procesos pueden ejecutar en paralelo. Además, múltiples hilos del mismo proceso pueden estar asignados a distintos procesadores y pueden ejecutar de modo concurrente. Un proceso multihilo puede lograr la concurrencia sin la sobrecarga del uso de múltiples procesos. Los hilos del mismo proceso pueden intercambiar información a través de su espacio de direcciones común y tienen acceso a los recursos compartidos del proceso. Los hilos de diferentes procesos pueden intercambiar información a través del uso de memoria compartida.

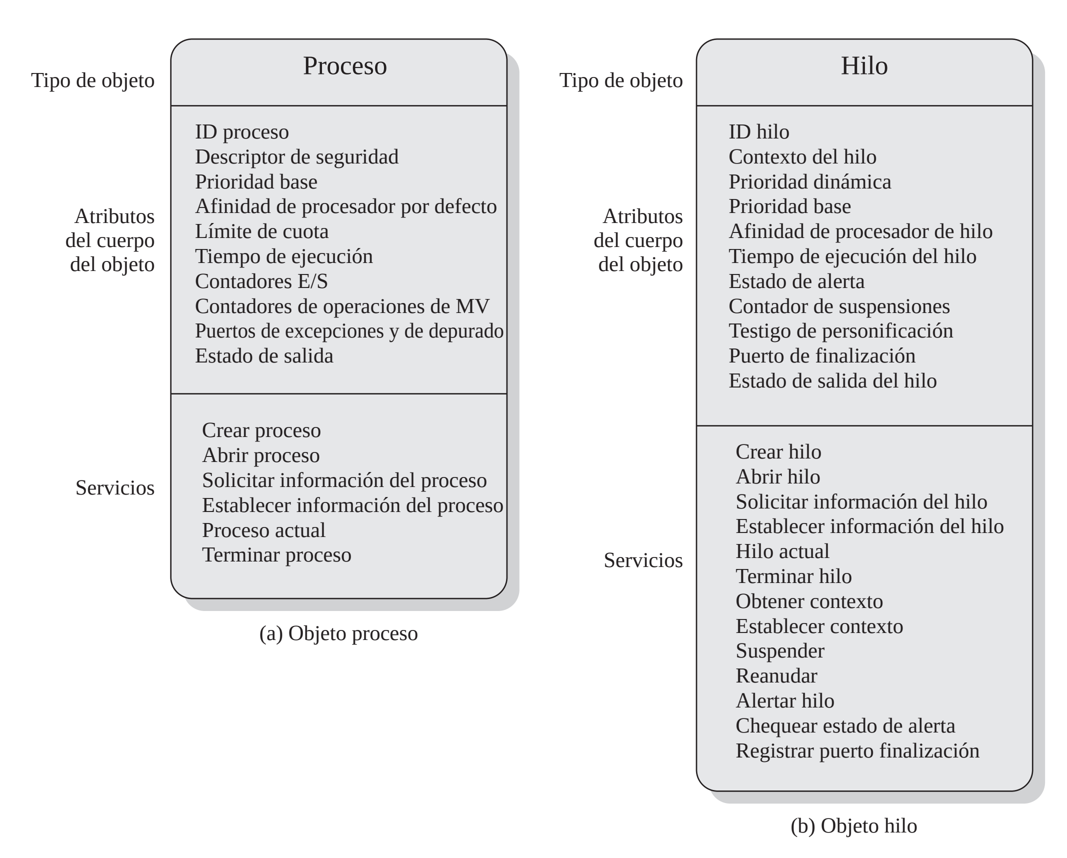


Figura 4.13. Objetos de Windows proceso e hilo.

Tabla 4.3. Atributos del objeto proceso de Windows.

ID proceso	Un valor único que identifica al proceso en el sistema operativo.
Descriptor de seguridad	Describe al creador del objeto, quién puede acceder o utilizar el objeto y a quién se le deniega acceso al objeto.
Prioridad base	Prioridad de ejecución base para los hilos del proceso.
Afinidad de procesador por defecto	Conjunto de procesadores por defecto, en los que pueden ejecutar los hilos del proceso.
Límite de cuota	Máxima cantidad de memoria del sistema paginada y no paginada, espacio del archivo de páginas y tiempo de procesador que pueden utilizar los procesos de un usuario.
Tiempo de ejecución	La cantidad total de tiempo que han ejecutado todos los hilos de un proceso.
Contadores de E/S	Variables que almacenan el número y tipo de operaciones de E/S que han realizado los hilos de un proceso.
Contadores de operaciones de MV	Variables que almacenan el número y tipo de operaciones de memoria virtual que han realizado los hilos de un proceso.
Puertos de excepciones y de depurado	Canales de comunicación entre procesos a los que el gestor de procesos manda un mensaje cuando uno de los hilos del proceso causa una excepción.
Estado de salida	La razón de la terminación del proceso.

Tabla 4.4. Atributos del objeto hilo de Windows.

ID hilo	Valor único que identifica a un hilo cuando llama a un servidor.
Contexto del hilo	El conjunto de los valores de los registros y otra información volátil que define el estado de ejecución de un hilo.
Prioridad dinámica	La prioridad de ejecución de un hilo en un determinado momento.
Prioridad base	El límite inferior de la prioridad dinámica de un hilo.
Afinidad de procesador asociada al hilo	El conjunto de procesadores en el que puede ejecutar un hilo, que es un subconjunto del valor definido en el objeto proceso.
Tiempo de ejecución del hilo	La cantidad de tiempo acumulado que ha ejecutado un hilo en modo usuario y modo núcleo.
Estado de alerta	Un flag que indica si el hilo debe ejecutar una llamada a procedimiento asíncrona.
Contador de suspensión	El número de veces que ha sido suspendida la ejecución de un hilo sin ser reanudado.
Testigo de personificación	Una señal de acceso temporal que permite al hilo realizar operaciones en lugar de otro proceso (utilizado por los subsistemas).
Puerto de finalización	Canal de comunicación entre procesos al que el gestor de procesos manda un mensaje cuando termina el hilo (utilizado por los subsistemas).
Estado de salida del hilo	La razón de la terminación del hilo.

Un proceso multihilo orientado a objetos es una forma efectiva de implementar una aplicación servidora. Por ejemplo, un proceso servidor podría atender a varios clientes. Cada petición de cliente desencadena la creación de un nuevo hilo del servidor.

ESTADO DE LOS HILOS

Un hilo de Windows se encuentra en uno de estos seis estados (Figura 4.14):

- **Listo (*ready*).** Puede planificarse para ejecución. El activador del micronúcleo conoce todos los hilos listos y los planifica en orden de prioridad.
- **Substituto (*standby*).** Un hilo substituto se ha seleccionado para ejecutar en siguiente lugar en un determinado procesador. Si la prioridad del hilo substituto es suficientemente alta, el hilo actualmente en ejecución en ese procesador podría ser expulsado en su favor. De otra forma, el hilo substituto espera hasta que el hilo en ejecución se bloquea o finaliza su porción de tiempo.
- **Ejecutando (*running*).** Una vez que el micronúcleo realiza un intercambio de hilo o proceso, el hilo susbtituto pasa al estado de ejecución y ejecuta hasta que es expulsado, finaliza su porción de tiempo, se bloquea o termina. En los dos primeros casos vuelve a la cola de listos.
- **Esperando (*waiting*).** Un hilo pasa a estado esperando cuando (1) se bloquea en un evento (por ejemplo, E/S), (2) espera voluntariamente por temas de sincronización, o (3) un subsistema manda al hilo a estado de suspendido. Cuando se satisface la condición de espera, el hilo pasa al estado Listo si todos sus recursos están disponibles.

- **Transición (*transition*).** Un hilo entra en este estado después de esperar si está listo para ejecutar pero los recursos no están disponibles. Por ejemplo, la pila del hilo puede no estar en memoria. Cuando los recursos están disponibles, el hilo pasa al estado Listo.
- **Terminado (*terminated*).** Un hilo se puede finalizar por sí mismo, por otro hilo o cuando su proceso padre finaliza. Cuando se completan las tareas internas, el hilo se borra del sistema, o puede retenerse por el *ejecutivo*[8] para futuras reinicializaciones.

SOPORTE PARA SUBSISTEMAS DE SISTEMAS OPERATIVOS

Los servicios de procesos e hilos de propósito general, deben dar soporte a las estructuras de procesos e hilos de varios SO cliente. Cada subsistema de SO es responsable de sacar provecho de los procesos e hilos de Windows para su propio sistema operativo. Esta área de gestión de procesos/hilos es complicada, y nosotros sólo damos una pequeña visión general.

La creación de un proceso comienza con la petición de una aplicación de un nuevo proceso. La aplicación manda una solicitud de creación de proceso a su correspondiente subsistema, que pasa la solicitud al ejecutivo de Windows. El ejecutivo crea un objeto proceso y devuelve al subsistema el manejador de dicho objeto. Cuando Windows crea un proceso, no crea automáticamente un hilo. En el caso de Win32 y OS/2, siempre se crea un nuevo proceso con un hilo. Por consiguiente, para estos sistemas operativos, el subsistema llama de nuevo al gestor de procesos de Windows para crear un hilo para el nuevo proceso, recibiendo un manejador de hilo como respuesta. A continuación se devuelven a la aplicación la información del hilo y del proceso. En el caso de Windows 16-bit y POSIX,

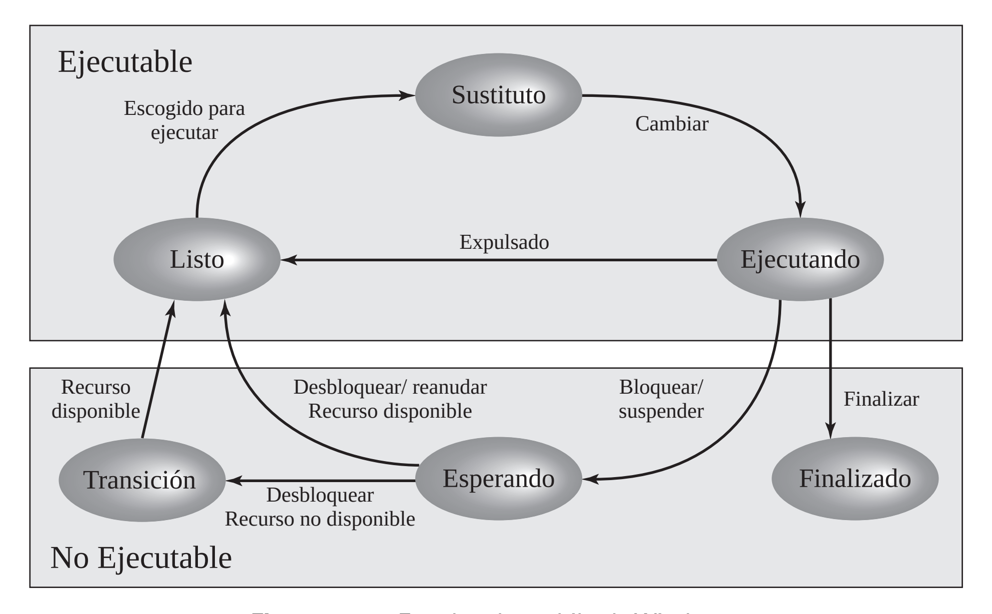


Figura 4.14. Estados de un hilo de Windows.

[8] El ejecutivo de Windows se describe en el Capítulo 2. Contiene los servicios base del sistema operativo, como gestión de memoria, gestión de procesos e hilos, seguridad, E/S y comunicación entre procesos.

no se soportan los hilos. Por tanto, para estos sistemas operativos, el subsistema obtiene un hilo para el nuevo proceso de Windows, para que el proceso pueda activarse, pero devuelve sólo la información del proceso a la aplicación. El hecho de que el proceso de la aplicación esté implementado como un hilo, no es visible para la aplicación.

Cuando se crea un nuevo proceso en Win32 o OS/2, el nuevo proceso hereda muchos de sus atributos del proceso que le ha creado. Sin embargo, en el entorno Windows, este procedimiento de creación se realiza indirectamente. El proceso cliente de una aplicación manda su solicitud de creación de proceso al subsistema del SO; un proceso del subsistema a su vez manda una solicitud de proceso al ejecutivo de Windows. Ya que el efecto deseado es que el nuevo proceso herede las características del proceso cliente, y no del proceso servidor, Windows permite al subsistema especificar el padre del nuevo proceso. El nuevo proceso hereda el testigo de acceso, límite de cuota, prioridad base y afinidad a procesador por defecto de su padre.

SOPORTE PARA MULTIPROCESAMIENTO SIMÉTRICO

Windows soporta una configuración hardware SMP. Los hilos de cualquier proceso, incluyendo los del ejecutivo, pueden ejecutar en cualquier procesador. En ausencia de restricciones de afinidad, explicadas en el siguiente párrafo, el micronúcleo asigna un hilo listo al siguiente procesador disponible. Esto asegura que ningún procesador está ocioso o está ejecutando un hilo de menor prioridad cuando un hilo de mayor prioridad está listo. Múltiples hilos de un proceso pueden ejecutar a la vez en múltiples procesadores.

Por defecto, el micronúcleo utiliza la política **afinidad débil** (*soft affinity*) para asignar procesadores a los hilos: el planificador intenta asignar un proceso listo al mismo procesador que lo ejecutó la última vez. Esto ayuda a reutilizar datos que estén todavía en la memoria cache del procesador de la ejecución previa del hilo. Para una aplicación es posible restringir la ejecución de sus hilos a determinados procesadores **afinidad fuerte** (*hard affinity*).

4.5. GESTIÓN DE HILOS Y SMP EN SOLARIS

Solaris implementa un soporte de hilo multinivel poco habitual, diseñado para proporcionar considerable flexibilidad para sacar provecho de los recursos del procesador.

ARQUITECTURA MULTIHILO

Solaris utiliza cuatro conceptos relacionados con los hilos:

- **Procesos.** Es un proceso normal UNIX e incluye el espacio de direcciones del usuario, la pila y el bloque de control del proceso.
- **Hilos de nivel de usuario.** Implementados a través de una biblioteca de hilos en el espacio de direcciones de un proceso, son invisibles al sistema operativo. Los hilos de nivel de usuario (*user-level threads*, ULT)[9] son la interfaz para las aplicaciones paralelas.

[9] De nuevo, el acrónimo ULT es exclusivo de este libro y no se encuentra en la literatura de Solaris.

- **Procesos ligeros.** Un proceso ligero (lightweight process, LWP) puede ser visto como una asociación entre ULT e hilos de núcleo. Cada LWP soporta uno o más ULT y se asocia con un hilo de núcleo. Los LWP se planifican de forma independiente por el núcleo y pueden ejecutar en paralelo en múltiples procesadores.
- **Hilos de núcleo.** Son entidades fundamentales que se pueden planificar para ejecutar en cualquier procesador del sistema.

La Figura 4.15 muestra la relación entre estas cuatro entidades. Nótese que hay siempre un hilo de núcleo por cada LWP. Un LWP es visible dentro de un proceso de la aplicación. De esta forma, las estructuras de datos LWP existen dentro del espacio de direcciones del proceso respectivo. Al mismo tiempo, cada LWP está vinculado a un único hilo de núcleo activable, y la estructura de datos para ese hilo de núcleo se mantiene dentro del espacio de direcciones del núcleo.

En nuestro ejemplo, el proceso 1 consiste en un único ULT vinculado con un único LWP. De esta forma, hay un único hilo de ejecución, correspondiente a un proceso UNIX tradicional. Cuando no se requiere concurrencia en un proceso, una aplicación utiliza esta estructura de proceso. El proceso 2 se corresponde con una estrategia ULT pura. Todos los ULT están soportados por un único hilo de núcleo, y por tanto sólo se puede ejecutar un ULT al mismo tiempo. Esta estructura es útil para una aplicación que se puede programar de forma que exprese la concurrencia, pero que no es necesario ejecutar en paralelo con múltiples hilos. El proceso 3 muestra múltiples hilos multiplexados en un menor número de LWP. En general, Solaris permite a las aplicaciones multiplexar ULT en un número menor o igual de LWP. Esto permite a la aplicación especificar el grado de paralelismo a nivel de núcleo que tendrá este proceso. El proceso 4 tiene sus hilos permanentemente vinculados a LWP de uno en uno. Esta estructura hace el paralelismo a nivel de núcleo totalmente visible a la aplicación. Es útil si los

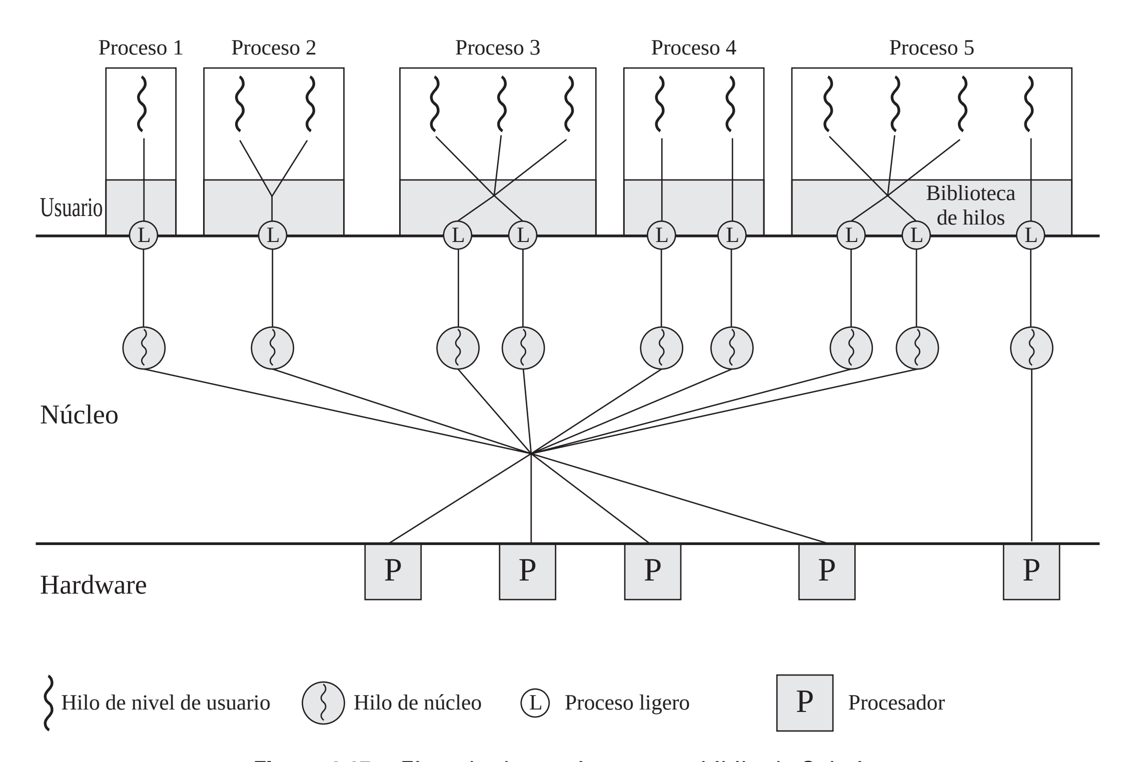


Figura 4.15. Ejemplo de arquitectura multihilo de Solaris.

hilos van a suspenderse frecuentemente al quedar bloqueados. El proceso 5 muestra tanto la asociación de múltiples ULT en múltiples LWP como el enlace de un ULT con un LWP. Además, hay un LWP asociado a un procesador particular.

Lo que no se muestra en la figura, es la presencia de hilos de núcleo que no están asociados con LWP. El núcleo crea, ejecuta y destruye estos hilos de núcleo para ejecutar funciones específicas del sistema. El uso de hilos de núcleo en lugar de procesos de núcleo para implementar funciones del sistema reduce la sobrecarga de los intercambios dentro del núcleo (de cambios de procesos a cambios de hilos).

MOTIVACIÓN

La combinación de los hilos a nivel de usuario y a nivel de núcleo, le da la oportunidad al programador de la aplicación de explotar la concurrencia de la forma más efectiva y apropiada para su aplicación.

Algunos programas tienen un paralelismo lógico del que se puede sacar provecho para simplificar y estructurar el código, pero que no necesitan paralelismo hardware. Por ejemplo, una aplicación que emplee múltiples ventanas, de las cuales sólo una está activa en un determinado momento, podría beneficiarse de la implementación como un conjunto de ULT en un único LWP. La ventaja de restringir estas aplicaciones a ULT es la eficiencia. Los ULT se pueden crear, destruir, bloquear, activar, etc., sin involucrar al núcleo. Si el núcleo conociera cada ULT, tendría que asignar estructuras de datos de núcleo para cada uno de ellos y realizar el intercambio de hilos. Como hemos visto (Tabla 4.1), el intercambio de hilos de nivel de núcleo es más costoso que el intercambio de hilos de nivel de usuario.

Si una aplicación tiene hilos que se pueden bloquear, por ejemplo cuando realiza E/S, resultará interesante tener múltiples LWP para soportar un número igual o mayor de ULT. Ni la aplicación ni la biblioteca de hilos necesitan hacer contorsiones para permitir ejecutar a otros hilos del mismo proceso. En su lugar, si un hilo de un proceso se bloquea, otros hilos del mismo proceso pueden ejecutar en los restantes LWP.

Asociar ULT y LWP uno a uno, es efectivo para algunas aplicaciones. Por ejemplo, una computación paralela de matrices podría dividir las filas de sus matrices en diferentes hilos. Si hay exactamente un ULT por LWP, no se requiere ningún intercambio de hilos en el proceso de computación.

La mezcla de hilos que están permanentemente asociados con LWP e hilos no asociados (múltiples hilos compartiendo múltiples LWP) es apropiada en algunas aplicaciones. Por ejemplo, una aplicación de tiempo real podría querer que algunos hilos tuvieran una gran prioridad en el sistema y planificación en tiempo real, mientras que otros hilos realizan funciones secundarias y pueden compartir un pequeño conjunto de LWP.

ESTRUCTURA DE LOS PROCESOS

La Figura 4.16 compara, en términos generales, la estructura de un proceso de un sistema UNIX tradicional con la de Solaris. En una implementación UNIX típica, la estructura del proceso incluye el ID del proceso; el ID del usuario; una tabla de tratamiento de señales, que usa el núcleo para decidir qué hacer cuando llega una señal a un proceso; descriptores de archivos, que describen el estado de los archivos en uso por el proceso; un mapa de memoria, que define el espacio de direcciones de este proceso; y una estructura del estado del procesador, que incluye la pila del núcleo para este proceso. Solaris contiene esta estructura básica pero reemplaza el bloque de estado del procesador con una lista de estructuras que contienen un bloque de datos por cada LWP.

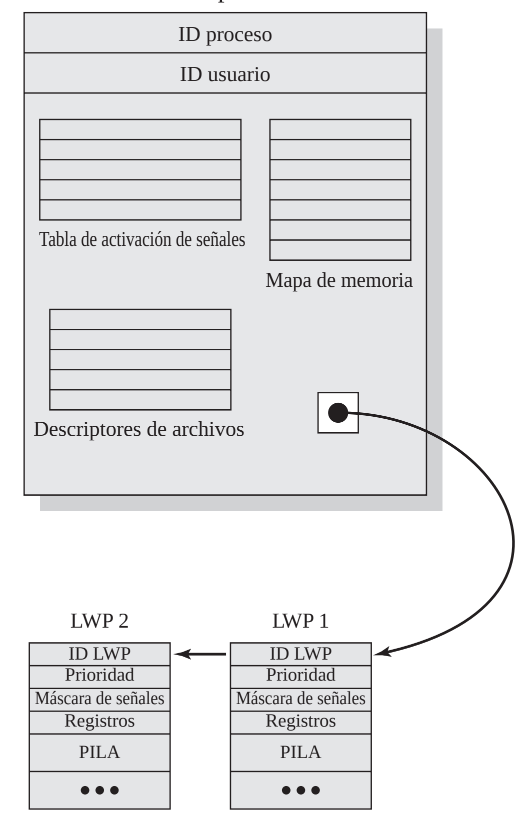


Figura 4.16. Estructura de procesos en UNIX tradicional y Solaris.

La estructura de datos de LWP incluye los siguientes elementos:

- Un identificador LWP.
- La prioridad de este LWP y, por tanto, del hilo del núcleo que le da soporte.
- Una máscara de señales que dice al núcleo qué señales se aceptarán.
- Valores almacenados de los registros a nivel de usuario (cuando el LWP no está ejecutando).
- La pila del núcleo para este LWP. Que incluye argumentos de llamadas al sistema, resultados y códigos de error para cada nivel de llamada.
- Uso de recursos y datos de perfiles.
- Puntero al correspondiente hilo del núcleo.
- Puntero a la estructura del proceso.

EJECUCIÓN DE HILOS

La Figura 4.17 muestra una visión simplificada de los estados de ejecución de ULT y LWP. La ejecución de los hilos a nivel de usuario se gestiona por la biblioteca de los hilos. Consideremos primero

los hilos no vinculados, es decir, los hilos que comparten una serie de LWP. Un hilo no vinculado puede estar en uno de los siguientes estados: ejecutable, activo, durmiendo o detenido. Un ULT en un estado activo se asigna a un LWP y ejecuta mientras que el hilo de núcleo subyacente ejecuta. Hay una serie de eventos que pueden causar que el ULT deje el estado activo. Consideremos un ULT activo denominado T1. Pueden ocurrir los siguientes sucesos:

- **Sincronización.** T1 invoca una de las primitivas concurrentes discutidas en el Capítulo 5 para coordinar su actividad con otros hilos y para forzar exclusión mutua. T1 se pasa a estado durmiendo. Cuando se cumple la condición de sincronización, T1 se pasa a estado ejecutable.
- **Suspensión.** Cualquier hilo (incluyendo T1) puede causar que se suspenda T1 y que se pase a estado detenido. T1 permanece en este estado hasta que otro hilo manda una petición de continuación, que lo pasa a estado ejecutable.
- **Expulsión.** Un hilo activo (T1 o cualquier otro hilo) hace algo que causa que otro hilo (T2) de mayor prioridad pase a ejecutable. Si T1 es el hilo con la menor prioridad, se expulsa y se pasa a estado ejecutable, y T2 se asigna al LWP disponible.
- **Ceder paso.** Si T1 ejecuta el mandato de biblioteca `thr_yield()`, el planificador de hilos de la biblioteca mirará si hay otro proceso ejecutable (T2) de la misma prioridad. Si existe, T1 se pasa a estado ejecutable y T2 se asigna al LWP liberado. Si no existe, T1 continúa ejecutando.

En todos los casos precedentes, cuando T1 sale del estado activo, la biblioteca de hilos selecciona otro hilo no vinculado en estado ejecutable y lo ejecuta en el LWP que está nuevamente disponible.

La Figura 4.17 también muestra el diagrama de estados de un LWP. Podemos ver este diagrama de estados como una descripción detallada del estado activo del ULT, ya que un hilo no vinculado sólo tiene un LWP asignado cuando está en estado Activo. El diagrama de estados de LWP se explica por sí mismo. Un hilo activo sólo está ejecutando cuando su LWP está en el estado Ejecutando. Cuando un hilo activo ejecuta una llamada al sistema bloqueante, el LWP pasa a estado Bloqueado. Sin embargo, el ULT permanece vinculado a ese LWP y, hasta donde concierne a la biblioteca de hilos, el ULT permanece activo.

Con los hilos vinculados, la relación entre ULT y LWP es ligeramente diferente. Por ejemplo, si un ULT vinculado pasa al estado Durmiendo mientras espera un evento de sincronización, su LWP también debe parar de ejecutar. Esto se logra manteniendo el bloqueo de LWP en una variable de sincronización a nivel de núcleo.

INTERRUPCIONES COMO HILOS

La mayor parte de los sistemas operativos contienen dos formas de actividad concurrente: procesos e interrupciones. Los procesos (o hilos) cooperan entre sí y gestionan el uso de estructuras de datos compartidas a través de diversas primitivas que fuerzan la exclusión mutua (sólo un proceso al mismo tiempo puede ejecutar determinado código o acceder a determinados datos) y que sincronizan su ejecución. Las interrupciones se sincronizan impidiendo su manejo por un periodo de tiempo. Solaris unifica estos dos conceptos en un solo modelo. Para hacer esto, las interrupciones se convierten a hilos de núcleo.

La motivación para convertir las interrupciones en hilos, es reducir la sobrecarga. El proceso de tratamiento de las interrupciones, a menudo maneja datos compartidos con el resto del núcleo. Por consiguiente, mientras que se ejecuta una rutina de núcleo que accede a esos datos, las interrupciones

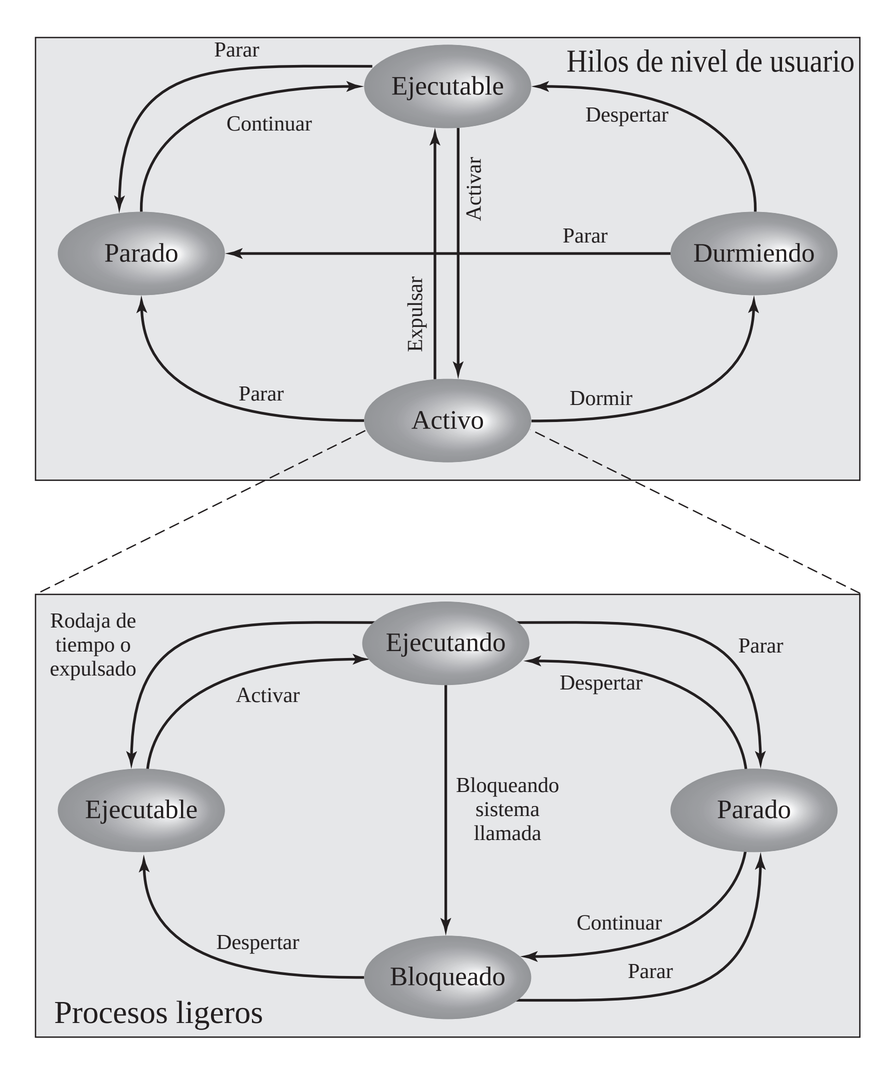


Figura 4.17. Estados de los hilos de nivel de usuarios y LWP en Solaris.

se deben bloquear, incluso aunque la mayor parte de las interrupciones no vayan a afectar a esos datos. Normalmente, la forma de hacer esto es que la rutina suba el nivel de prioridad de interrupciones, para bloquear las interrupciones, y baje el nivel de prioridad cuando el acceso está completado. Estas operaciones llevan tiempo. El problema es mayor en un sistema multiprocesador. El núcleo debe proteger más objetos y podría necesitar bloquear las interrupciones en todos los procesadores.

La solución de Solaris se puede resumir de la siguiente manera:

1. Solaris emplea un conjunto de hilos de núcleo para manejar las interrupciones. Como cualquier hilo del núcleo, un hilo de interrupción tiene su propio identificador, prioridad, contexto y pila.

2. El núcleo controla el acceso a las estructuras de datos y sincroniza los hilos de interrupción utilizando primitivas de exclusión mutua, del tipo discutido en el Capítulo 5. Es decir, se utilizan las técnicas habituales de sincronización de hilos en el manejo de interrupciones.

3. Se asigna mayor prioridad a los hilos de interrupción que a cualquier otro tipo de hilo de núcleo.

Cuando sucede una interrupción, se envía a un determinado procesador y el hilo que estuviera ejecutando en ese procesador es inmovilizado (*pinned*). Un hilo inmovilizado no se puede mover a otro procesador y su contexto se preserva; simplemente se suspende hasta que se procesa la interrupción. Entonces, el procesador comienza a ejecutar un hilo de interrupción. Hay un conjunto de hilos de interrupción desactivados disponibles, por lo que no es necesario crear un nuevo hilo. A continuación ejecuta el hilo de interrupción para manejar la interrupción. Si la rutina de interrupción necesita acceso a una estructura de memoria que ya está bloqueada por otro hilo en ejecución, el hilo de interrupción deberá esperar para acceder a dicha estructura. Un hilo de interrupción sólo puede ser expulsado por otro hilo de interrupción de mayor prioridad.

La experimentación con las interrupciones de Solaris indican que este enfoque proporciona un rendimiento superior a las estrategias de manejo de interrupciones tradicionales [KLEI95].

4.6. GESTIÓN DE PROCESOS E HILOS EN LINUX

TAREAS LINUX

Un proceso, o tarea, en Linux se representa por una estructura de datos `task_struct`, que contiene información de diversas categorías:

- **Estado.** El estado de ejecución del proceso (ejecutando, listo, suspendido, detenido, *zombie*). Pasaremos a describirlo posteriormente.
- **Información de planificación.** Información necesitada por Linux para planificar procesos. Un proceso puede ser normal o de tiempo real y tener una prioridad. Los procesos de tiempo real se planifican antes que los procesos normales y, dentro de cada categoría, se pueden usar prioridades relativas. Hay un contador que lleva la cuenta de la cantidad de tiempo que un proceso ha estado ejecutando.
- **Identificadores.** Cada proceso tiene un identificador único de proceso y también tiene identificadores de usuario y grupo. Un identificador de usuario se utiliza para asignar privilegios de acceso a recursos a un grupo de procesos.
- **Comunicación entre procesos.** Linux soporta el mecanismo IPC encontrado en UNIX SVR4, descrito en el Capítulo 6.
- **Enlaces.** Cada proceso incluye un enlace a sus padres, enlaces a sus hermanos (procesos con el mismo padre), y enlaces a todos sus hijos.
- **Tiempos y temporizadores.** Incluye el tiempo de creación del proceso y la cantidad de tiempo de procesador consumido por el proceso hasta el momento. Un proceso también puede tener asociado uno o más temporizadores. Un proceso define un temporizador a través de una llamada al sistema; como resultado se manda una señal al proceso cuando finaliza el temporizador. Un temporizador puede ser de un solo uso o periódico.
- **Sistema de archivos.** Incluye punteros a cualquier archivo abierto por este proceso, así como punteros a los directorios actual y raíz para este proceso.
- **Espacio de direcciones.** Define el espacio de direcciones virtual asignado a este proceso.

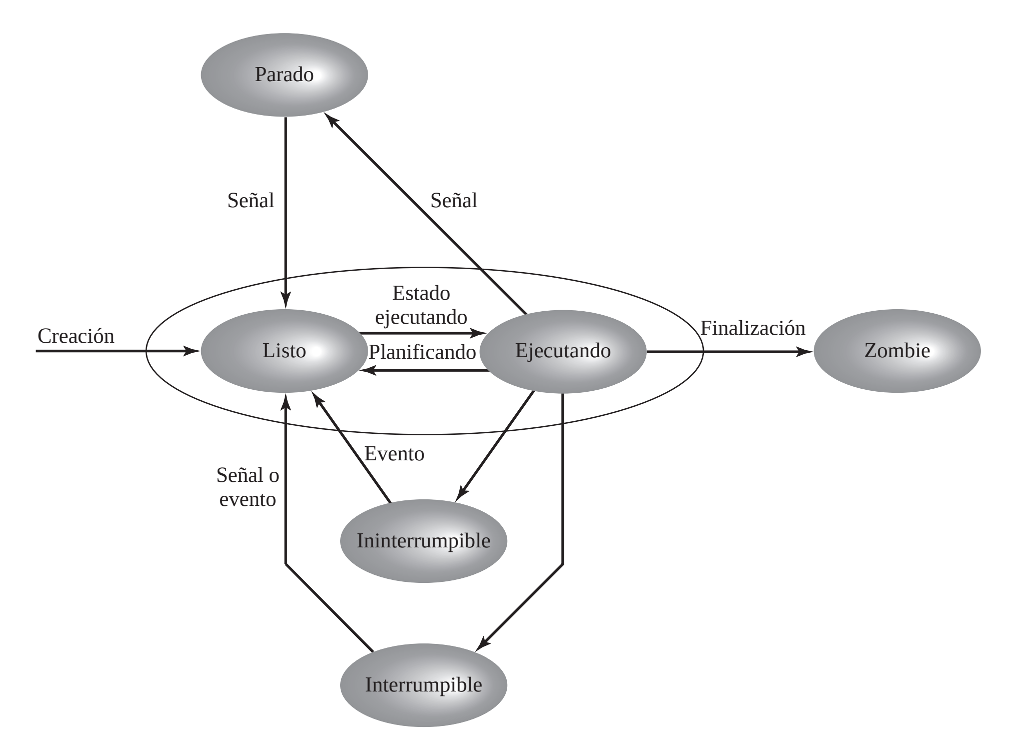


Figura 4.18. Modelo de procesos e hilos en Linux.

- **Contexto específico del procesador.** La información de los registros y de la pila que constituyen el contexto de este proceso.
- **Ejecutando.** Este valor de estado se corresponde con dos estados. Un proceso Ejecutando puede estar ejecutando o está listo para ejecutar.
- **Interrumpible.** Es un estado bloqueado, en el que el proceso está esperando por un evento, tal como la finalización de una operación de E/S, la disponibilidad de un recurso o una señal de otro proceso.
- **Ininterrumpible.** Éste es otro estado bloqueado. La diferencia entre este estado y el estado Interrumpible es que en el estado Ininterrumpible un proceso está esperando directamente sobre un estado del hardware y por tanto no manejará ninguna señal.
- **Detenido.** El proceso ha sido parado y sólo puede ser reanudado por la acción positiva de otro proceso. Por ejemplo, un proceso que está siendo depurado se puede poner en estado Parado.
- ***Zombie*.** El proceso se ha terminado pero, por alguna razón, todavía debe tener su estructura de tarea en la tabla de procesos.

HILOS LINUX

Los sistemas UNIX tradicionales soportan un solo hilo de ejecución por proceso, mientras que los sistemas UNIX modernos suelen proporcionar soporte para múltiples hilos de nivel de núcleo por proceso. Como con los sistemas UNIX tradicionales, las versiones antiguas del núcleo de Linux no ofrecían soporte multihilo. En su lugar, las aplicaciones debían escribirse con un conjunto de fun-

ciones de biblioteca de nivel de usuario. La más popular de estas bibliotecas se conoce como *biblioteca pthread (POSIX thread)*, en donde se asociaban todos los hilos en un único proceso a nivel de núcleo. Hemos visto que las versiones modernas de UNIX ofrecen hilos a nivel de núcleo. Linux proporciona una solución única en la que no diferencia entre hilos y procesos. Utilizando un mecanismo similar al de los procesos ligeros de Solaris, los hilos de nivel de usuario se asocian con procesos de nivel de núcleo. Múltiples hilos de nivel de usuario que constituyen un único proceso de nivel de usuario, se asocian con procesos Linux a nivel de núcleo y comparten el mismo ID de grupo. Esto permite a estos procesos compartir recursos tales como archivos y memorias y evitar la necesidad de un cambio de contexto cuando el planificador cambia entre procesos del mismo grupo.

En Linux se crea un nuevo proceso copiando los atributos del proceso actual. Un nuevo proceso se puede *clonar* de forma que comparte sus recursos, tales como archivos, manejadores de señales y memoria virtual. Cuando los dos procesos comparten la misma memoria virtual, funcionan como hilos de un solo proceso. Sin embargo, no se define ningún tipo de estructura de datos independiente para un hilo. En lugar del mandato normal `fork()`, los procesos se crean en Linux usando el mandato `clone()`. Este mandato incluye un conjunto de flags como argumentos, definidos en la Tabla 4.5. La llamada al sistema tradicional `fork()` se implementa en Linux con la llamada al sistema `clone()` sin ningún flag.

Cuando el núcleo de Linux realiza un cambio de un proceso a otro, verifica si la dirección del directorio de páginas del proceso actual es la misma que en el proceso a ser planificado. Si lo es, están compartiendo el mismo espacio de direcciones, por lo que el cambio de contexto consiste básicamente en saltar de una posición del código a otra.

Aunque los procesos clonados que son parte del mismo grupo de procesos, pueden compartir el mismo espacio de memoria, no pueden compartir la misma pila de usuario. Por tanto, la llamada `clone()` crea espacios de pila separados para cada proceso.

Tabla 4.5. Flags de la llamada `clone()` de Linux.

CLONE_CLEARID	Borrar el ID de tarea.
CLONE_DETACHED	El padre no quiere el envío de la señal SIGCHLD en su finalización.
CLONE_FILES	Compartir la tabla que identifica los archivos abiertos.
CLONE_FS	Compartir la tabla que identifica al directorio raíz y al directorio actual de trabajo, así como el valor de la máscara de bits utilizada para enmascarar los permisos iniciales de un nuevo archivo.
CLONE_IDLETASK	Establecer el PID a cero, que se refiere a la tarea *idle*. La tarea *idle* se utiliza cuando todas las tareas disponibles están bloqueadas esperando por recursos.
CLONE_NEWNS	Crear un nuevo espacio de nombres para el hijo.
CLONE_PARENT	El llamante y la nueva tarea comparten el mismo proceso padre.
CLONE_PTRACE	Si el proceso padre está siendo trazado, el proceso hijo también lo hará.
CLONE_SETTID	Escribir el TID en el espacio de usuario.
CLONE_SETTLS	Crear un nuevo TLS para el hijo.
CLONE_SIGHAND	Compartir la tabla que identifica los manejadores de señales.

CLONE_SYSVSEM	Compartir la semántica SEM_UNDO de System V.
CLONE_THREAD	Insertar este proceso en el mismo grupo de hilos del padre. Si este flag está activado, fuerza de forma implícita a CLONE_PARENT.
CLONE_VFORK	Si está activado, el padre no se planifica para ejecución hasta que el hijo invoque la llamada al sistema *execve()*.
CLONE_VM	Compartir el espacio de direcciones (descriptor de memoria y todas las tablas de páginas).

4.7. RESUMEN

Algunos sistemas operativos distinguen los conceptos de proceso e hilo, el primero relacionado con la propiedad de recursos y el segundo relacionado con la ejecución de programas. Este enfoque podría llevar a mejorar la eficiencia y la conveniencia del código. En un sistema multihilo, se pueden definir múltiples hilos concurrentes en un solo proceso. Esto se podría hacer utilizando tanto hilos de nivel de usuario como hilos de nivel de núcleo. El sistema operativo desconce la existencia de los hilos de nivel de usuario y se crean y gestionan por medio de una biblioteca de hilos que ejecuta en el espacio de usuario de un proceso. Los hilos de nivel de usuario son muy eficientes por que no se requiere ningún cambio de contexto para cambiar de uno a otro hilo. Sin embargo, sólo puede estar ejecutando al mismo tiempo un único hilo de nivel de usuario, y si un hilo se bloquea, el proceso entero hará lo mismo. Los hilos a nivel de núcleo son hilos de un proceso que se mantienen en el núcleo. Como son reconocidos por el núcleo, múltiples hilos del mismo proceso pueden ejecutar en paralelo en un multiprocesador y el bloqueo de un hilo no bloquea al proceso completo. Sin embargo, se requiere un cambio de contexto para cambiar de un hilo a otro.

El multiprocesamiento simétrico es un método de organizar un sistema multiprocesador de tal forma que cualquier proceso (o hilo) puede ejecutar en cualquier procesador; esto incluye al código del núcleo y a los procesos. Las arquitecturas SMP generan nuevos conceptos de diseño en los sistemas operativos y proporcionan mayor rendimiento que un sistema uniprocesador bajo las mismas condiciones.

En los últimos años, ha habido mucho interés en el diseño de los sistemas operativos basados en micronúcleo. En su forma pura, un sistema operativo micronúcleo consiste en un micronúcleo muy pequeño que ejecuta en modo núcleo y que sólo contiene las funciones más esenciales y críticas del sistema operativo. El resto de funciones del sistema operativo se implementa para ejecutar en modo usuario y para utilizar el micronúcleo para servicios críticos. El diseño de tipo micronúcleo lleva a implementaciones flexibles y altamente modulares. Sin embargo, todavía persisten algunas preguntas sobre el rendimiento de estas arquitecturas.

4.8. LECTURAS RECOMENDADAS

[LEWI96] y [KLEI96] proporcionan una buena visión general del concepto de los hilos y una discusión sobre las estrategias de programación; el primero se centra más en conceptos y el último se centra más en programación, pero ambos proporcionan una buena cobertura de ambos temas. [PHAM96] trata los hilos de Windows NT en profundidad. [ROBB04] tiene una buena cobertura sobre el concepto de los hilos en UNIX.

[MUKH96] proporciona una buena discusión sobre los aspectos de diseño de los sistemas operativos para SMP. [CHAP97] contiene cinco artículos sobre las direcciones del diseño actual de los sistemas operativos multiprocesador. [LIED95] y [LIED96] contienen discusiones interesantes de los principios del diseño de micronúcleos; el último se centra en aspectos del rendimiento.

CHAP97 Chapin, S., y Maccabe, A., eds. «Multiprocessor Operating Systems: Harnessing the Power.» Special issue of *IEEE Concurrency*, Abril-Junio 1997.

KLEI96 Kleiman, S.; Shah, D.; y Smallders, B. *Programming with Threads.* Upper Saddle River, NJ: Prentice Hall, 1996.

LEWI96 Lewis, B., y Berg, D. *Threads Primer.* Upper Saddle River, NJ: Prentice Hall, 1996.

LIED95 Liedtke, J. «On μ-Kernel Construction.» *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Diciembre 1995.

LIED96 Liedtke, J. «Toward Real Microkernels.» *Communications of the ACM*, Septiembre 1996.

MUKH96 Mukherjee, B., y Karsten, S. «Operating Systems for Parallel Machines.» in *Parallel Computers: Theory y Practice*. Edited by T. Casavant, P. Tvrkik, y F. Plasil. Los Alamitos, CA: IEEE Computer Society Press, 1996.

PHAM96 Pham, T., y Garg, P. *Multithreaded Programming with Windows NT.* Upper Saddle River, NJ: Prentice Hall, 1996.

ROBB04 Robbins, K., y Robbins, S. *UNIX Systems Programming: Communication, Concurrency, y Threads.* Upper Saddle River, NJ: Prentice Hall, 2004.

4.9. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

hilo	micronúcleo	procesos ligeros
hilos de nivel de núcleo (KLT)	multihilo	puerto
hilo de nivel de usuario (ULT)	multiprocesador simétrico (SMP)	sistema operativo monolítico
mensaje	proceso	tarea

PREGUNTAS DE REVISIÓN

4.1. La Tabla 3.5 enumera los elementos típicos que se encuentran en un bloque de control de proceso para un sistema operativo monohilo. De éstos, ¿cuáles deben pertenecer a un bloque de control de hilo y cuáles deben pertenecer a un bloque de control de proceso para un sistema multihilo?

4.2. Enumere las razones por las que un cambio de contexto entre hilos puede ser más barato que un cambio de contexto entre procesos.

4.3. ¿Cuáles son las dos características diferentes y potencialmente independientes en el concepto de proceso?

4.4. Dé cuatro ejemplos generales del uso de hilos en un sistema multiprocesador monousuario.

4.5. ¿Qué recursos son compartidos normalmente por todos los hilos de un proceso?

4.6. Enumere tres ventajas de los ULT sobre los KLT.

4.7. Enumere dos desventajas de los ULT en comparación con los KLT.

4.8. Defina *jacketing (revestimiento)*.

4.9. Defina brevemente las diversas arquitecturas que se nombran en la Figura 4.8.

4.10. Enumere los aspectos principales de diseño de un sistema operativo SMP.

4.11. Dé ejemplos de servicios y funciones que se encuentran en un sistema operativo monolítico típico que podrían ser subsistemas externos en un sistema operativo micronúcleo.

4.12. Enumere y explique brevemente siete ventajas potenciales de un diseño micronúcleo en comparación con un diseño monolítico.

4.13. Explique la desventaja potencial de rendimiento de un sistema operativo micronúcleo.

4.14. Enumere cuatro funciones que le gustaría encontrar incluso en un sistema operativo micronúcleo mínimo.

4.15. ¿Cuál es la forma básica de comunicación entre procesos o hilos en un sistema operativo micronúcleo?

PROBLEMAS

4.1. Se dijo que las dos ventajas de utilizar múltiples hilos dentro de un proceso son que (1) se necesita menos trabajo en la creación de un nuevo hilo dentro de un proceso existente que en la creación de un nuevo proceso, y (2) se simplifica la comunicación entre hilos del mismo proceso. ¿Es cierto también que el cambio de contexto entre dos hilos del mismo proceso conlleva menos trabajo que el cambio de contexto entre dos hilos de diferentes procesos?

4.2. En la discusión de ULT frente a KLT, se señaló que una desventaja de los ULT es que cuando ejecutan una llamada al sistema, no sólo se bloquea ese hilo, sino todos los hilos del proceso. ¿Por qué sucede esto?

4.3. En OS/2, lo que normalmente abarca el concepto de proceso en otros sistemas operativos, se divide en tres tipos de entidad: sesión, procesos e hilos. Una sesión es una colección de uno o más hilos asociados con el interfaz de usuario (teclado, pantalla y ratón). La sesión representa una aplicación de usuario interactiva, tal como un procesador de textos o una hoja de cálculo. Este concepto permite al usuario abrir más de una aplicación, cada una de ellas con una o más ventanas en pantalla. El sistema operativo debe saber qué ventana, y por tanto qué sesión, está activa, de forma que las entradas del teclado y del ratón se manden a la sesión apropiada. En todo momento, hay una sesión en primer plano, y el resto de las sesiones está en segundo plano. Todas las entradas del teclado y del ratón se redirigen a uno de los procesos de la sesión en primer plano, según lo indique la aplicación. Cuando una sesión está en primer plano, un proceso que realiza salida gráfica, manda la señal directamente al *buffer* del hardware gráfico y, por tanto, a la pantalla. Cuando se pasa la sesión a segundo plano, se almacena el *buffer* del hardware gráfico en un *buffer* lógico de vídeo para esa sesión. Mientras que una sesión está en segundo plano, si cualquiera de los hilos de cualquiera de los procesos de esa sesión ejecuta y produce alguna salida por pantalla, esa salida se redirige al *buffer* lógico de vídeo. Cuando la sesión vuelve a primer plano, se actualiza la pantalla para reflejar el contenido del *buffer* lógico de vídeo.

Hay una forma de reducir el número de conceptos relacionados con los procesos en OS/2 de tres a dos: eliminar las sesiones y asociar el interfaz de usuario (teclado, pantalla y ra-

tón) con procesos. De esta forma, sólo hay un proceso en primer plano. Para mayor estructuración, los procesos pueden estar formados por hilos.

a) ¿Qué beneficios se pierden en este enfoque?

b) Si utiliza estas modificaciones, ¿dónde asigna recursos (memoria, archivos, etc.): a nivel de proceso o de hilo?

4.4. Considérese un entorno en el que hay una asociación uno a uno entre hilos de nivel de usuario e hilos de nivel de núcleo que permite a uno o más hilos de un proceso ejecutar llamadas al sistema bloqueantes mientras otros hilos continúan ejecutando. Explique por qué este modelo puede hacer que ejecuten más rápido los programas multihilo que sus correspondientes versiones monohilo en una máquina monoprocesador.

4.5. Si un proceso termina y hay todavía ejecutando hilos de ese proceso, ¿continuarán ejecutando?

4.6. El sistema operativo OS/390 para *mainframe* se estructura sobre los conceptos de espacio de direcciones y tarea. Hablando de forma aproximada, un espacio de direcciones único corresponde a una única aplicación y corresponde más o menos a un proceso en otros sistemas operativos. Dentro de un espacio de direcciones, varias tareas se pueden generar y ejecutar de forma concurrente; esto corresponde, más o menos, al concepto de multihilo. Hay dos estructuras de datos claves para gestionar esta estructura de tareas. Un bloque de control del espacio de direcciones (ASCB) contiene información sobre un espacio de direcciones, y es requerido por el OS/390 esté o no ejecutando ese espacio de direcciones. La información del ASBC incluye la prioridad de la planificación, memoria real y virtual asignada a ese espacio de direcciones, el número de tareas listas en ese espacio de direcciones, y si cada una de ellas está en la zona de intercambio. Un bloque de control de tareas (TCB) representa a un programa de usuario en ejecución. Contiene información necesaria para gestionar una tarea dentro de un espacio de direcciones e incluye información del estado del procesador, punteros a programas que son parte de esta tarea y el estado de ejecución de la tarea. Los ASCB son estructuras globales mantenidas en la memoria del sistema, mientras que los TCB son estructuras locales que se mantienen en su espacio de direcciones. ¿Cuál es la ventaja de dividir la información de control en porciones globales y locales?

4.7. Un multiprocesador con ocho procesadores tiene 20 unidades de cinta. Hay un gran número de trabajos enviados al sistema, cada uno de los cuales requiere un máximo de cuatro cintas para completar su ejecución. Suponga que cada trabajo comienza ejecutando con sólo 3 unidades de cinta durante largo tiempo, antes de requerir la cuarta cinta durante un periodo breve cerca de su finalización. Suponga también un número infinito de estos trabajos.

a) Suponga que el planificador del sistema operativo no comenzará un trabajo a menos que haya cuatro cintas disponibles. Cuando comienza un trabajo, se asignan cuatro cintas de forma inmediata y no se liberan hasta que el trabajo finaliza. ¿Cuál es el número máximo de trabajos que pueden estar en progreso en un determinado momento? ¿Cuál es el número máximo y mínimo de unidades de cinta que pueden estar paradas como resultado de esta política?

b) Sugiera una política alternativa para mejorar la utilización de las unidades de cinta y al mismo tiempo evitar un interbloqueo en el sistema. ¿Cuál es el número máximo de trabajos que pueden estar en progreso en un determinado momento? ¿Cuáles son los límites del número de unidades de cinta paradas?

4.8. En la descripción de los estados de un ULT en Solaris, se dijo que un ULT podría ceder el paso a otro ULT de la misma prioridad. ¿No sería posible que hubiera un hilo ejecutable de mayor prioridad y que, por tanto, la función de cesión diera como resultado la cesión a un hilo de la misma o de mayor prioridad?

CAPÍTULO 5

Concurrencia. Exclusión mutua y sincronización

Los temas centrales del diseño de sistemas operativos están todos relacionados con la gestión de procesos e hilos:

- **Multiprogramación.** Gestión de múltiples procesos dentro de un sistema monoprocesador.
- **Multiprocesamiento.** Gestión de múltiples procesos dentro de un multiprocesador.
- **Procesamiento distribuido.** Gestión de múltiples procesos que ejecutan sobre múltiples sistemas de cómputo distribuidos.

La concurrencia es fundamental en todas estas áreas y en el diseño del sistema operativo. La concurrencia abarca varios aspectos, entre los cuales están la comunicación entre procesos y la compartición de, o competencia por, recursos, la sincronización de actividades de múltiples procesos y la reserva de tiempo de procesador para los procesos. Debemos entender que todos estos asuntos no sólo suceden en el entorno del multiprocesamiento y el procesamiento distribuido, sino también en sistemas monoprocesador multiprogramados.

La concurrencia aparece en tres contextos diferentes:

- **Múltiples aplicaciones.** La multiprogramación fue ideada para permitir compartir dinámicamente el tiempo de procesamiento entre varias aplicaciones activas.
- **Aplicaciones estructuradas.** Como extensión de los principios del diseño modular y de la programación estructurada, algunas aplicaciones pueden ser programadas eficazmente como un conjunto de procesos concurrentes.
- **Estructura del sistema operativo.** Las mismas ventajas constructivas son aplicables a la programación de sistemas y, de hecho, los sistemas operativos son a menudo implementados en sí mismos como un conjunto de procesos o hilos.

Dada la importancia de este tema, cuatro de los capítulos de este libro tratan aspectos relacionados con la concurrencia. Este capítulo y el siguiente tratan de la concurrencia en sistemas multiprogramados y con multiproceso. Los Capítulos 13 y 14 examinan aspectos de la concurrencia en relación con el procesamiento distribuido. Aunque el resto de este libro cubre cierto número de otros temas importantes del diseño del sistema operativo, en nuestra opinión la concurrencia juega un papel principal frente a todos estos otros temas.

Este capítulo comienza con una introducción al concepto de concurrencia y a las implicaciones de la ejecución de múltiples procesos concurrentes[1]. Se descubre que el requisito básico para conseguir ofrecer procesos concurrentes es la capacidad de hacer imperar la exclusión mutua; esto es, la capacidad de impedir a cualquier proceso realizar una acción mientras se le haya permitido a otro. Después, se examinan algunos mecanismos hardware que pueden permitir conseguir exclusión mutua. Entonces, se verán soluciones que no requieren espera activa y que pueden ser tanto ofrecidas por el sistema operativo como impuestas por el compilador del lenguaje. Se examinan tres propuestas: semáforos, monitores y paso de mensajes.

Se utilizan dos problemas clásicos de concurrencia para ilustrar los conceptos y comparar las propuestas presentadas en este capítulo. El problema productor/consumidor se introduce pronto para utilizarse luego como ejemplo. El capítulo se cierra con el problema de los lectores/escritores.

---

[1] Por simplicidad, generalmente nos referiremos a la ejecución concurrente de *procesos*. De hecho, como se ha visto en el capítulo anterior, en algunos sistemas la unidad fundamental de concurrencia es el hilo en vez del proceso.

Tabla 5.1. Algunos términos clave relacionados con la concurrencia.

sección crítica (***critical section***)	Sección de código dentro de un proceso que requiere acceso a recursos compartidos y que no puede ser ejecutada mientras otro proceso esté en una sección de código correspondiente.
interbloqueo (***deadlock***)	Situación en la cual dos o más procesos son incapaces de actuar porque cada uno está esperando que alguno de los otros haga algo.
círculo vicioso (***livelock***)	Situación en la cual dos o más procesos cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.
exclusión mutua (***mutual exclusion***)	Requisito de que cuando un proceso esté en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.
condición de carrera (***race condition***)	Situación en la cual múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.
inanición (***starvation***)	Situación en la cual un proceso preparado para avanzar es soslayado indefinidamente por el planificador; aunque es capaz de avanzar, nunca se le escoge.

La exposición sobre la concurrencia continúa en el Capítulo 6 y la discusión de los mecanismos de concurrencia de nuestros sistemas de ejemplo se pospone hasta el final del capítulo.

La Tabla 5.1 muestra algunos términos clave relacionados con la concurrencia.

5.1. PRINCIPIOS DE LA CONCURRENCIA

En un sistema multiprogramado de procesador único, los procesos se entrelazan en el tiempo para ofrecer la apariencia de ejecución simultánea (Figura 2.12a). Aunque no se consigue procesamiento paralelo real, e ir cambiando de un proceso a otro supone cierta sobrecarga, la ejecución entrelazada proporciona importantes beneficios en la eficiencia del procesamiento y en la estructuración de los programas. En un sistema de múltiples procesadores no sólo es posible entrelazar la ejecución de múltiples procesos sino también solaparlas (Figura 2.12b).

En primera instancia, puede parecer que el entrelazado y el solapamiento representan modos de ejecución fundamentalmente diferentes y que presentan diferentes problemas. De hecho, ambas técnicas pueden verse como ejemplos de procesamiento concurrente y ambas presentan los mismos problemas. En el caso de un monoprocesador, los problemas surgen de una característica básica de los sistemas multiprogramados: no puede predecirse la velocidad relativa de ejecución de los procesos. Ésta depende de la actividad de los otros procesos, de la forma en que el sistema operativo maneja las interrupciones y de las políticas de planificación del sistema operativo. Se plantean las siguientes dificultades:

1. La compartición de recursos globales está cargada de peligros. Por ejemplo, si dos procesos utilizan ambos la misma variable global y ambos realizan lecturas y escrituras sobre esa variable, entonces el orden en que se ejecuten las lecturas y escrituras es crítico. En la siguiente subsección se muestra un ejemplo de este problema.
2. Para el sistema operativo es complicado gestionar la asignación de recursos de manera óptima. Por ejemplo, el proceso A puede solicitar el uso de un canal concreto de E/S, y serle concedido el control, y luego ser suspendido justo antes de utilizar ese canal. Puede no ser desea-

ble que el sistema operativo simplemente bloquee el canal e impida su utilización por otros procesos; de hecho esto puede conducir a una condición de interbloqueo, tal como se describe en el Capítulo 6.

3. Llega a ser muy complicado localizar errores de programación porque los resultados son típicamente no deterministas y no reproducibles (por ejemplo, *véase* en [LEBL87, CARR89, SHEN02] discusiones sobre este asunto).

Todas las dificultades precedentes se presentan también en un sistema multiprocesador, porque aquí tampoco es predecible la velocidad relativa de ejecución de los procesos. Un sistema multiprocesador debe bregar también con problemas derivados de la ejecución simultánea de múltiples procesos. Sin embargo, fundamentalmente los problemas son los mismos que aquéllos de un sistema monoprocesador. Esto quedará claro a medida que la exposición avance.

UN EJEMPLO SENCILLO

Considere el siguiente procedimiento:

```
void eco ()
{
        cent = getchar();
        csal = cent;
        putchar(csal);
}
```

Este procedimiento muestra los elementos esenciales de un programa que proporcionará un procedimiento de eco de un carácter; la entrada se obtiene del teclado, una tecla cada vez. Cada carácter introducido se almacena en la variable `cent`. Luego se transfiere a la variable `csal` y se envía a la pantalla. Cualquier programa puede llamar repetidamente a este procedimiento para aceptar la entrada del usuario y mostrarla por su pantalla.

Considere ahora que se tiene un sistema multiprogramado de un único procesador y para un único usuario. El usuario puede saltar de una aplicación a otra, y cada aplicación utiliza el mismo teclado para la entrada y la misma pantalla para la salida. Dado que cada aplicación necesita usar el procedimiento `eco`, tiene sentido que éste sea un procedimiento compartido que esté cargado en una porción de memoria global para todas las aplicaciones. De este modo, sólo se utiliza una única copia del procedimiento `eco`, economizándose espacio.

La compartición de memoria principal entre procesos es útil para permitir una interacción eficiente y próxima entre los procesos. No obstante, esta interacción puede acarrear problemas. Considere la siguiente secuencia:

1. El proceso P1 invoca el procedimiento `eco` y es interrumpido inmediatamente después de que `getchar` devuelva su valor y sea almacenado en `cent`. En este punto, el carácter introducido más recientemente, *x*, está almacenado en `cent`.

2. El proceso P2 se activa e invoca al procedimiento `eco`, que ejecuta hasta concluir, habiendo leído y mostrado en pantalla un único carácter, *y*.

3. Se retoma el proceso P1. En este instante, el valor *x* ha sido sobrescrito en `cent` y por tanto se ha perdido. En su lugar, `cent` contiene *y*, que es transferido a `csal` y mostrado.

Así, el primer carácter se pierde y el segundo se muestra dos veces. La esencia de este problema es la variable compartida global, `cent.` Múltiples procesos tienen acceso a esta variable. Si un proceso actualiza la variable global y justo entonces es interrumpido, otro proceso puede alterar la variable antes de que el primer proceso pueda utilizar su valor. Suponga ahora, que decidimos que sólo un proceso al tiempo pueda estar en dicho procedimiento. Entonces la secuencia anterior resultaría como sigue:

1. El proceso P1 invoca el procedimiento `eco` y es interrumpido inmediatamente después de que concluya la función de entrada. En este punto, el carácter introducido más recientemente, *x*, está almacenado en `cent`.
2. El proceso P2 se activa e invoca al procedimiento `eco.` Sin embargo, dado que P1 está todavía dentro del procedimiento `eco,` aunque actualmente suspendido, a P2 se le impide entrar en el procedimiento. Por tanto, P2 se suspende esperando la disponibilidad del procedimiento `eco.`
3. En algún momento posterior, el proceso P1 se retoma y completa la ejecución de **eco**. Se muestra el carácter correcto, *x*.
4. Cuando P1 sale de `eco,` esto elimina el bloqueo de P2. Cuando P2 sea más tarde retomado, invocará satisfactoriamente el procedimiento `eco`.

La lección a aprender de este ejemplo es que es necesario proteger las variables globales compartidas (así como otros recursos globales compartidos) y que la única manera de hacerlo es controlar el código que accede a la variable. Si imponemos la disciplina de que sólo un proceso al tiempo pueda entrar en `eco` y que una vez en `eco` el procedimiento debe ejecutar hasta completarse antes de estar disponible para otro proceso, entonces el tipo de error que se acaba de describir no ocurrirá. Cómo se puede imponer esa disciplina es uno de los temas capitales de este capítulo.

Este problema fue enunciado con la suposición de que se trataba de un sistema operativo multiprogramado para un monoprocesador. El ejemplo demuestra que los problemas de la concurrencia suceden incluso cuando hay un único procesador. En un sistema multiprocesador, aparecen los mismos problemas de recursos compartidos protegidos, y funcionan las mismas soluciones. Primero, supóngase que no hay mecanismo para controlar los accesos a la variable global compartida:

1. Los procesos P1 y P2 están ambos ejecutando, cada cual en un procesador distinto. Ambos procesos invocan el procedimiento `eco`.
2. Ocurren los siguientes eventos; los eventos en la misma línea suceden en paralelo.

Proceso P1	**Proceso P2**
•	•
cent = getchar();	•
•	cent = getchar();
csal = cent;	csal = cent;
putchar(csal);	•
•	putchar(csal);
•	•

El resultado es que el carácter introducido a P1 se pierde antes de ser mostrado, y el carácter introducido a P2 es mostrado por ambos P1 y P2. De nuevo, añádase la capacidad de cumplir la

disciplina de que sólo un proceso al tiempo pueda estar en `eco`. Entonces, sucede la siguiente secuencia:

1. Los procesos P1 y P2 están ambos ejecutando, cada cual en un procesador distinto. Ambos procesos invocan al procedimiento `eco`.
2. Mientras P1 está dentro del procedimiento `eco`, P2 invoca a `eco`. Dado que P1 está todavía dentro del procedimiento `eco` (ya esté P1 suspendido o ejecutando), a P2 se le bloqueará la entrada al procedimiento. Por tanto, P2 se suspende en espera de la disponibilidad del procedimiento `eco`.
3. En algún momento posterior, el proceso P1 completa la ejecución de `eco`, sale del procedimiento y continúa ejecutando. Inmediatamente después de que P1 salga de `eco`, se retoma P2 que comienza la ejecución de `eco`.

En el caso de un sistema monoprocesador, el motivo por el que se tiene un problema es que una interrupción puede parar la ejecución de instrucciones en cualquier punto de un proceso. En el caso de un sistema multiprocesador, se tiene el mismo motivo y, además, puede suceder porque dos procesos pueden estar ejecutando simultáneamente y ambos intentando acceder a la misma variable global. Sin embargo, la solución a ambos tipos de problema es la misma: controlar los accesos a los recursos compartidos.

CONDICIÓN DE CARRERA

Una condición de carrera sucede cuando múltiples procesos o hilos leen y escriben datos de manera que el resultado final depende del orden de ejecución de las instrucciones en los múltiples procesos. Consideremos dos casos sencillos.

Como primer ejemplo, suponga que dos procesos, P1 y P2, comparten la variable global `a`. En algún punto de su ejecución, P1 actualiza `a` al valor 1 y, en el mismo punto en su ejecución, P2 actualiza `a` al valor 2. Así, las dos tareas compiten en una carrera por escribir la variable `a`. En este ejemplo el «perdedor» de la carrera (el proceso que actualiza el último) determina el valor de `a`.

Para nuestro segundo ejemplo, considere dos procesos, P3 y P4, que comparten las variables globales `b` y `c`, con valores iniciales `b` = 1 y `c` = 2. En algún punto de su ejecución, P3 ejecuta la asignación `b = b + c` y, en algún punto de su ejecución, P4 ejecuta la asignación `c = b + c`. Note que los dos procesos actualizan diferentes variables. Sin embargo, los valores finales de las dos variables dependen del orden en que los dos procesos ejecuten estas dos asignaciones. Si P3 ejecuta su sentencia de asignación primero, entonces los valores finales serán `b` = 3 y `c` = 5. Si P4 ejecuta su sentencia de asignación primero, entonces los valores finales serán `b` = 4 y `c` = 3.

El Apéndice A trata sobre las condiciones de carrera, utilizando los semáforos como ejemplo.

PREOCUPACIONES DEL SISTEMA OPERATIVO

¿Qué aspectos de diseño y gestión surgen por la existencia de la concurrencia? Pueden enumerarse las siguientes necesidades:

1. El sistema operativo debe ser capaz de seguir la pista de varios procesos. Esto se consigue con el uso de bloques de control de proceso y fue descrito en el Capítulo 4.

2. El sistema operativo debe ubicar y desubicar varios recursos para cada proceso activo. Estos recursos incluyen:

    - **Tiempo de procesador.** Esta es la misión de la planificación, tratada en la Parte Cuatro.
    - **Memoria.** La mayoría de los sistemas operativos usan un esquema de memoria virtual. El tema es abordado en la Parte Tres.
    - **Ficheros.** Tratados en el Capítulo 12.
    - **Dispositivos de E/S.** Tratados en el Capítulo 11.

3. El sistema operativo debe proteger los datos y recursos físicos de cada proceso frente a interferencias involuntarias de otros procesos. Esto involucra técnicas que relacionan memoria, ficheros y dispositivos de E/S. En el Capítulo 15 se encuentra tratado en general el tema de la protección.
4. El funcionamiento de un proceso y el resultado que produzca, debe ser independiente de la velocidad a la que suceda su ejecución en relación con la velocidad de otros procesos concurrentes. Este es el tema de este capítulo.

Para entender cómo puede abordarse la cuestión de la independencia de la velocidad, necesitamos ver las formas en que los procesos pueden interaccionar.

INTERACCIÓN DE PROCESOS

Podemos clasificar las formas en que los procesos interaccionan en base al grado en que perciben la existencia de cada uno de los otros. La Tabla 5.2 enumera tres posibles grados de percepción más las consecuencias de cada uno:

- **Procesos que no se perciben entre sí.** Son procesos independientes que no se pretende que trabajen juntos. El mejor ejemplo de esta situación es la multiprogramación de múltiples procesos independientes. Estos bien pueden ser trabajos por lotes o bien sesiones interactivas o una mezcla. Aunque los procesos no estén trabajando juntos, el sistema operativo necesita preocuparse de la **competencia** por recursos. Por ejemplo, dos aplicaciones independientes pueden querer ambas acceder al mismo disco, fichero o impresora. El sistema operativo debe regular estos accesos.
- **Procesos que se perciben indirectamente entre sí.** Son procesos que no están necesariamente al tanto de la presencia de los demás mediante sus respectivos ID de proceso, pero que comparten accesos a algún objeto, como un *buffer* de E/S. Tales procesos exhiben **cooperación** en la compartición del objeto común.
- **Procesos que se perciben directamente entre sí.** Son procesos capaces de comunicarse entre sí vía el ID del proceso y que son diseñados para trabajar conjuntamente en cierta actividad. De nuevo, tales procesos exhiben **cooperación**.

Las condiciones no serán siempre tan claras como se sugiere en la Tabla 5.2. Mejor dicho, algunos procesos pueden exhibir aspectos tanto de competición como de cooperación. No obstante, es constructivo examinar cada uno de los tres casos de la lista precedente y determinar sus implicaciones para el sistema operativo.

Tabla 5.2. Interacción de procesos.

Grado de percepción	Relación	Influencia que un proceso tiene sobre el otro	Potenciales problemas de control
Procesos que no se perciben entre sí	Competencia	• Los resultados de un proceso son independientes de la acción de los otros La temporización del proceso puede verse afectada	• Exclusión mutua • Interbloqueo (recurso renovable) • Inanición
Procesos que se perciben indirectamente entre sí (por ejemplo, objeto compartido)	Cooperación por compartición	• Los resultados de un proceso pueden depender de la información obtenida de otros • La temporización del proceso puede verse afectada	• Exclusión mutua • Interbloqueo (recurso renovable) • Inanición • Coherencia de datos
Procesos que se perciben directamente entre sí (tienen primitivas de comunicación a su disposición)	Cooperación por comunicación	• Los resultados de un proceso pueden depender de la información obtenida de otros • La temporización del proceso puede verse afectada	• Interbloqueo (recurso consumible) • Inanición

Competencia entre procesos por recursos. Los procesos concurrentes entran en conflicto entre ellos cuando compiten por el uso del mismo recurso. En su forma pura, puede describirse la situación como sigue. Dos o más procesos necesitan acceso a un recurso durante el curso de su ejecución. Ningún proceso se percata de la existencia de los otros procesos y ninguno debe verse afectado por la ejecución de los otros procesos. Esto conlleva que cada proceso debe dejar inalterado el estado de cada recurso que utilice. Ejemplos de recursos son los dispositivos de E/S, la memoria, el tiempo de procesador y el reloj.

No hay intercambio de información entre los procesos en competencia. No obstante, la ejecución de un proceso puede afectar al comportamiento de los procesos en competencia. En concreto, si dos procesos desean ambos acceder al mismo recurso único, entonces, el sistema operativo reservará el recurso para uno de ellos, y el otro tendrá que esperar. Por tanto, el proceso al que se le deniega el acceso será ralentizado. En un caso extremo, el proceso bloqueado puede no conseguir nunca el recurso y por tanto no terminar nunca satisfactoriamente.

En el caso de procesos en competencia, deben afrontarse tres problemas de control. Primero está la necesidad de **exclusión mutua**. Supóngase que dos o más procesos requieren acceso a un recurso único no compartible, como una impresora. Durante el curso de la ejecución, cada proceso estará enviando mandatos al dispositivo de E/S, recibiendo información de estado, enviando datos o recibiendo datos. Nos referiremos a tal recurso como un **recurso crítico**, y a la porción del programa que lo utiliza como la **sección crítica** del programa. Es importante que sólo se permita un programa al tiempo en su sección crítica. No podemos simplemente delegar en el sistema operativo para que entienda y aplique esta restricción porque los detalles de los requisitos pueden no ser obvios. En el caso de una impresora, por ejemplo, queremos que cualquier proceso individual tenga el control de

la impresora mientras imprime el fichero completo. De otro modo, las líneas de los procesos en competencia se intercalarían.

La aplicación de la exclusión mutua crea dos problemas de control adicionales. Uno es el del **interbloqueo**. Por ejemplo, considere dos procesos, P1 y P2, y dos recursos, R1 y R2. Suponga que cada proceso necesita acceder a ambos recursos para realizar parte de su función. Entonces es posible encontrarse la siguiente situación: el sistema operativo asigna R1 a P2, y R2 a P1. Cada proceso está esperando por uno de los dos recursos. Ninguno liberará el recurso que ya posee hasta haber conseguido el otro recurso y realizado la función que requiere ambos recursos. Los dos procesos están interbloqueados.

Un último problema de control es la **inanición**. Suponga que tres procesos (P1, P2, P3) requieren todos accesos periódicos al recurso R. Considere la situación en la cual P1 está en posesión del recurso y P2 y P3 están ambos retenidos, esperando por ese recurso. Cuando P1 termine su sección crítica, debería permitírsele acceso a R a P2 o P3. Asúmase que el sistema operativo le concede acceso a P3 y que P1 solicita acceso otra vez antes de completar su sección crítica. Si el sistema operativo le concede acceso a P1 después de que P3 haya terminado, y posteriormente concede alternativamente acceso a P1 y a P3, entonces a P2 puede denegársele indefinidamente el acceso al recurso, aunque no suceda un interbloqueo.

El control de la competencia involucra inevitablemente al sistema operativo ya que es quien ubica los recursos. Además, los procesos necesitarán ser capaces por sí mismos de expresar de alguna manera el requisito de exclusión mutua, como bloqueando el recurso antes de usarlo. Cualquier solución involucrará algún apoyo del sistema operativo, como proporcionar un servicio de bloqueo. La Figura 5.1 ilustra el mecanismo de exclusión mutua en términos abstractos. Hay *n* procesos para ser ejecutados concurrentemente. Cada proceso incluye (1) una sección crítica que opera sobre algún recurso Ra, y (2) código adicional que precede y sucede a la sección crítica y que no involucra acceso a Ra. Dado que todos los procesos acceden al mismo recurso Ra, se desea que sólo un proceso esté en su sección crítica al mismo tiempo. Para aplicar exclusión mutua se proporcionan dos funciones: `entrarcritica` y `salircritica`. Cada función toma como un argumento el nombre del recurso sujeto de la competencia. A cualquier proceso que intente entrar en su sección crítica mientras otro proceso está en su sección crítica, por el mismo recurso, se le hace esperar.

```
/* PROCESO 1 */
void P1
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

```
/* PROCESO 2 */
void P2
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

• • •

```
/* PROCESO n */
void Pn
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

Figura 5.1. Ilustración de la exclusión mutua.

Falta examinar mecanismos específicos para proporcionar las funciones `entrarcritica` y `salircritica`. Por el momento, postergamos este asunto mientras consideramos los otros casos de interacción de procesos.

Cooperación entre procesos vía compartición. El caso de cooperación vía compartición cubre procesos que interaccionan con otros procesos sin tener conocimiento explícito de ellos. Por ejemplo, múltiples procesos pueden tener acceso a variables compartidas o a ficheros o bases de datos compartidas. Los procesos pueden usar y actualizar los datos compartidos sin referenciar otros procesos pero saben que otros procesos pueden tener acceso a los mismos datos. Así, los procesos deben cooperar para asegurar que los datos que comparten son manipulados adecuadamente. Los mecanismos de control deben asegurar la integridad de los datos compartidos.

Dado que los datos están contenidos en recursos (dispositivos, memoria), los problemas de control de exclusión mutua, interbloqueo e inanición están presentes de nuevo. La única diferencia es que los datos individuales pueden ser accedidos de dos maneras diferentes, lectura y escritura, y sólo las operaciones de escritura deben ser mutuamente exclusivas.

Sin embargo, por encima de estos problemas, surge un nuevo requisito: el de la coherencia de datos. Como ejemplo sencillo, considérese una aplicación de contabilidad en la que pueden ser actualizados varios datos individuales. Supóngase que dos datos individuales a y b han de ser mantenidos en la relación $a = b$. Esto es, cualquier programa que actualice un valor, debe también actualizar el otro para mantener la relación. Considérense ahora los siguientes dos procesos:

P1:

$a = a + 1;$

$b = b + 1;$

P2:

$b = 2 * b;$

$a = 2 * a;$

Si el estado es inicialmente consistente, cada proceso tomado por separado dejará los datos compartidos en un estado consistente. Ahora considere la siguiente ejecución concurrente en la cual los dos procesos respetan la exclusión mutua sobre cada dato individual (a y b).

$a = a + 1;$

$b = 2 * b;$

$b = b + 1;$

$a = 2 * a;$

Al final de la ejecución de esta secuencia, la condición $a = b$ ya no se mantiene. Por ejemplo, si se comienza con $a = b = 1$, al final de la ejecución de esta secuencia, tendremos $a = 4$ y $b = 3$. El problema puede ser evitado declarando en cada proceso la secuencia completa como una sección crítica.

Así se ve que el concepto de sección crítica es importante en el caso de cooperación por compartición. Las mismas funciones abstractas `entrarcritica` y `salircritica` tratadas anteriormente (Figura 5.1) pueden usarse aquí. En este caso, el argumento de las funciones podría ser una variable, un fichero o cualquier otro objeto compartido. Es más, si las secciones críticas se utilizan para conse-

guir integridad de datos, entonces puede no haber recurso o variable concreta que pueda ser identificada como argumento. En este caso, puede pensarse en el argumento como un identificador compartido entre los procesos concurrentes que identifica las secciones críticas que deben ser mutuamente exclusivas.

Cooperación entre procesos vía comunicación. En los dos primeros casos que se han tratado, cada proceso tiene su propio entorno aislado que no incluye a los otros procesos. Las interacciones entre procesos son indirectas. En ambos casos hay cierta compartición. En el caso de la competencia, hay recursos compartidos sin ser conscientes de los otros procesos. En el segundo caso, hay compartición de valores y aunque cada proceso no es explícitamente consciente de los demás procesos, es consciente de la necesidad de mantener integridad de datos. Cuando los procesos cooperan vía comunicación, en cambio, los diversos procesos involucrados participan en un esfuerzo común que los vincula a todos ellos. La comunicación proporciona una manera de sincronizar o coordinar actividades varias.

Típicamente, la comunicación se fundamenta en mensajes de algún tipo. Las primitivas de envío y recepción de mensajes deben ser proporcionadas como parte del lenguaje de programación o por el núcleo del sistema operativo.

Dado que en el acto de pasar mensajes los procesos no comparten nada, la exclusión mutua no es un requisito de control en este tipo de cooperación. Sin embargo, los problemas de interbloqueo e inanición están presentes. Como ejemplo de interbloqueo, dos procesos pueden estar bloqueados, cada uno esperando por una comunicación del otro. Como ejemplo de inanición, considérense tres procesos, P1, P2 y P3, que muestran el siguiente comportamiento. P1 está intentando repetidamente comunicar con P2 o con P3, y P2 y P3 están ambos intentando comunicar con P1. Podría suceder una secuencia en la cual P1 y P2 intercambiasen información repetidamente, mientras P3 está bloqueado esperando comunicación de P1. No hay interbloqueo porque P1 permanece activo, pero P3 pasa hambre.

REQUISITOS PARA LA EXCLUSIÓN MUTUA

Cualquier mecanismo o técnica que vaya a proporcionar exclusión mutua debería cumplimentar los siguientes requisitos:

1. La exclusión mutua debe hacerse cumplir: sólo se permite un proceso al tiempo dentro de su sección crítica, de entre todos los procesos que tienen secciones críticas para el mismo recurso u objeto compartido.
2. Un proceso que se pare en su sección no crítica debe hacerlo sin interferir con otros procesos.
3. No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente: ni interbloqueo ni inanición.
4. Cuando ningún proceso esté en una sección crítica, a cualquier proceso que solicite entrar en su sección crítica debe permitírsele entrar sin demora.
5. No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
6. Un proceso permanece dentro de su sección crítica sólo por un tiempo finito.

Hay varias maneras de satisfacer los requisitos para la exclusión mutua. Una manera es delegar la responsabilidad en los procesos que desean ejecutar concurrentemente. Esos procesos, ya sean

programas del sistema o programas de aplicación, estarían obligados a coordinarse entre sí para cumplir la exclusión mutua, sin apoyo del lenguaje de programación ni del sistema operativo. Podemos referirnos a esto como soluciones software. Aunque este enfoque es propenso a una alta sobrecarga de procesamiento y a errores, sin duda es útil examinar estas propuestas para obtener una mejor comprensión de la complejidad de la programación concurrente. Este tema se cubre en el Apéndice A. Un segundo enfoque es proporcionar cierto nivel de soporte dentro del sistema operativo o del lenguaje de programación. Tres de los más importantes de estos enfoques se examinan en las Secciones 5.3 a 5.5.

5.2. EXCLUSIÓN MUTUA: SOPORTE HARDWARE

Se han desarrollado cierto número de algoritmos software para conseguir exclusión mutua, de los cuales el más conocido es el algoritmo de Dekker. La solución software es fácil que tenga una alta sobrecarga de procesamiento y es significativo el riesgo de errores lógicos. No obstante, el estudio de estos algoritmos ilustra muchos de los conceptos básicos y de los potenciales problemas del desarrollo de programas concurrentes. Para el lector interesado, el Apéndice A incluye un análisis de las soluciones software. En esta sección se consideran varias interesantes soluciones hardware a la exclusión mutua.

DESHABILITAR INTERRUPCIONES

En una máquina monoprocesador, los procesos concurrentes no pueden solaparse, sólo pueden entrelazarse. Es más, un proceso continuará ejecutando hasta que invoque un servicio del sistema operativo o hasta que sea interrumpido. Por tanto, para garantizar la exclusión mutua, basta con impedir que un proceso sea interrumpido. Esta técnica puede proporcionarse en forma de primitivas definidas por el núcleo del sistema para deshabilitar y habilitar las interrupciones. Un proceso puede cumplir la exclusión mutua del siguiente modo (compárese con la Figura 5.1):

```
while (true)
{
        /* deshabilitar interrupciones */;
        /* sección crítica */;
        /* habilitar interrupciones */;
        /* resto */;
}
```

Dado que la sección crítica no puede ser interrumpida, se garantiza la exclusión mutua. El precio de esta solución, no obstante, es alto. La eficiencia de ejecución podría degradarse notablemente porque se limita la capacidad del procesador de entrelazar programas. Un segundo problema es que esta solución no funcionará sobre una arquitectura multiprocesador. Cuando el sistema de cómputo incluye más de un procesador, es posible (y típico) que se estén ejecutando al tiempo más de un proceso. En este caso, deshabilitar interrupciones no garantiza exclusión mutua.

INSTRUCCIONES MÁQUINA ESPECIALES

En una configuración multiprocesador, varios procesadores comparten acceso a una memoria principal común. En este caso no hay una relación maestro/esclavo; en cambio los procesadores se compor-

tan independientemente en una relación de igualdad. No hay mecanismo de interrupción entre procesadores en el que pueda basarse la exclusión mutua.

A un nivel hardware, como se mencionó, el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición. Con este fundamento, los diseñadores de procesadores han propuesto varias instrucciones máquina que llevan a cabo dos acciones atómicamente[2], como leer y escribir o leer y comprobar, sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejecución de la instrucción, el acceso a la posición de memoria se le bloquea a toda otra instrucción que referencie esa posición. Típicamente, estas acciones se realizan en un único ciclo de instrucción.

En esta sección, se consideran dos de las instrucciones implementadas más comúnmente. Otras están descritas en [RAYN86] y [STON93].

Instrucción *Test and Set*. La instrucción *test and set* (comprueba y establece) puede definirse como sigue:

```
boolean testset (int i)
{
        if (i == 0)
        {
            i = 1;
            return true;
        }
        else
        {
            return false;
        }
}
```

La instrucción comprueba el valor de su argumento *i*. Si el valor es 0, entonces la instrucción reemplaza el valor por 1 y devuelve cierto. En caso contrario, el valor no se cambia y devuelve falso. La función `testset` completa se realiza atómicamente; esto es, no está sujeta a interrupción.

La Figura 5.2a muestra un protocolo de exclusión mutua basado en el uso de esta instrucción. La construcción `paralelos` (P1, P2, …, P*n*) significa lo siguiente: suspender la ejecución del programa principal; iniciar la ejecución concurrente de los procedimientos P1, P2, …, P*n*; cuando todos los P1, P2, …, P*n* hayan terminado, retomar al programa principal. Una variable compartida `cerrojo` se inicializa a 0. El único proceso que puede entrar en su sección crítica es aquél que encuentra la variable `cerrojo` igual a 0. Todos los otros procesos que intenten entrar en su sección crítica caen en un modo de espera activa. El término **espera activa** (*busy waiting*), o **espera cíclica** (*spin waiting*) se refiere a una técnica en la cual un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero continúa ejecutando una instrucción o conjunto de instrucciones que comprueban la variable apropiada para conseguir entrar. Cuando un proceso abandona su sección crítica, restablece `cerrojo` a 0; en este punto, a uno y sólo a uno de los procesos en espera se le concederá ac-

---

[2] El término atómico significa que la instrucción se realiza en un único paso y no puede ser interrumpida.

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(int i)
{
    while (true)
    {
        while (!testset (cerrojo))
            /* no hacer nada */;
        /* sección crítica */;
        cerrojo = 0;
        /* resto */
    }
}
void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), . . . ,P(n));
}
```

(a) Instrucción *test and set*

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(int i)
{
    int llavei = 1;
    while (true)
    {
        do exchange (llavei, cerrojo)
        while (llavei != 0);
        /* sección crítica */;
        exchange (llavei, cerrojo);
        /* resto */
    }
}
void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), . . ., P(n));
}
```

(b) Instrucción *exchange*

Figura 5.2. Soporte hardware para la exclusión mutua.

ceso a su sección crítica. La elección del proceso depende de cuál de los procesos es el siguiente que ejecuta la instrucción `testset`.

Instrucción *Exchange*. La instrucción *exchange* (intercambio) puede definirse como sigue:

```
void exchange (int registro, int memoria)
{
    int temp;
    temp = memoria;
    memoria = registro;
    registro = temp;
}
```

La instrucción intercambia los contenidos de un registro con los de una posición de memoria. Tanto la arquitectura Intel IA-32 (Pentium) como la IA-64 (Itanium) contienen una instrucción XCHG.

La Figura 5.2b muestra un protocolo de exclusión mutua basado en el uso de una instrucción *exchange*. Una variable compartida `cerrojo` se inicializa a 0. Cada proceso utiliza una variable local $\texttt{llave}_i$ que se inicializa a 1. El único proceso que puede entrar en su sección crítica es aquél que en-

cuentra `cerrojo` igual a 0, y al cambiar `cerrojo` a 1 se excluye a todos los otros procesos de la sección crítica. Cuando el proceso abandona su sección crítica, se restaura `cerrojo` al valor 0, permitiéndose que otro proceso gane acceso a su sección crítica.

Nótese que la siguiente expresión siempre se cumple dado el modo en que las variables son inicializadas y dada la naturaleza del algoritmo `exchange`:

$$cerrojo + \sum_i llave_i = n$$

Si `cerrojo` = 0, entonces ningún proceso está en su sección crítica. Si `cerrojo` = 1, entonces exactamente un proceso está en su sección crítica, aquél cuya variable $\texttt{llave}_\texttt{i}$ es igual a 0.

Propiedades de la solución instrucción máquina. El uso de una instrucción máquina especial para conseguir exclusión mutua tiene ciertas ventajas:

- Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria principal compartida.
- Es simple y, por tanto, fácil de verificar.
- Puede ser utilizado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida por su propia variable.

Hay algunas desventajas serias:

- **Se emplea espera activa.** Así, mientras un proceso está esperando para acceder a una sección crítica, continúa consumiendo tiempo de procesador.
- **Es posible la inanición.** Cuando un proceso abandona su sección crítica y hay más de un proceso esperando, la selección del proceso en espera es arbitraria. Así, a algún proceso podría denegársele indefinidamente el acceso.
- **Es posible el interbloqueo.** Considérese el siguiente escenario en un sistema de procesador único. El proceso P1 ejecuta la instrucción especial (por ejemplo, *testset*, *exchange*) y entra en su sección crítica. Entonces P1 es interrumpido para darle el procesador a P2, que tiene más alta prioridad. Si P2 intenta ahora utilizar el mismo recurso que P1, se le denegará el acceso, dado el mecanismo de exclusión mutua. Así caerá en un bucle de espera activa. Sin embargo, P1 nunca será escogido para ejecutar por ser de menor prioridad que otro proceso listo, P2.

Dados los inconvenientes de ambas soluciones software y hardware que se acaban de esbozar, es necesario buscar otros mecanismos.

5.3. SEMÁFOROS

Pasamos ahora a mecanismos del sistema operativo y del lenguaje de programación que se utilizan para proporcionar concurrencia. Comenzando, en esta sección, con los semáforos. Las siguientes dos secciones tratarán de monitores y de paso de mensajes.

El primer avance fundamental en el tratamiento de los problemas de programación concurrente ocurre en 1965 con el tratado de Dijkstra [DIJK65]. Dijkstra estaba involucrado en el diseño de un sistema operativo como una colección de procesos secuenciales cooperantes y con el desarrollo de

mecanismos eficientes y fiables para dar soporte a la cooperación. Estos mecanismos podrían ser usados fácilmente por los procesos de usuario si el procesador y el sistema operativo colaborasen en hacerlos disponibles.

El principio fundamental es éste: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Cualquier requisito complejo de coordinación puede ser satisfecho con la estructura de señales apropiada. Para la señalización, se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal vía el semáforo `s`, el proceso ejecutará la primitiva `semSignal(s)`. Para recibir una señal vía el semáforo ***s***, el proceso ejecutará la primitiva `semWait(s)`; si la correspondiente señal no se ha transmitido todavía, el proceso se suspenderá hasta que la transmisión tenga lugar[3].

Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual sólo están definidas tres operaciones:

1. Un semáforo puede ser inicializado a un valor no negativo.
2. La operación `semWait` decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando `semWait` se bloquea. En otro caso, el proceso continúa su ejecución.
3. La operación `semSignal` incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación `semWait`.

Aparte de estas tres operaciones no hay manera de inspeccionar o manipular un semáforo.

La Figura 5.3 sugiere una definición más formal de las primitivas del semáforo. Las primitivas `semWait` y `semSignal` se asumen atómicas. Una versión más restringida, conocida como **semáforo binario** o **mutex**, se define en la Figura 5.4. Un semáforo binario sólo puede tomar los valores 0 y 1 y se puede definir por las siguientes tres operaciones:

1. Un semáforo binario puede ser inicializado a 0 o 1.
2. La operación `semWaitB` comprueba el valor del semáforo. Si el valor es cero, entonces el proceso que está ejecutando `semWaitB` se bloquea. Si el valor es uno, entonces se cambia el valor a cero y el proceso continúa su ejecución.
3. La operación `semSignalB` comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en la operación `semWaitB`. Si no hay procesos bloqueados, entonces el valor del semáforo se pone a uno.

En principio debería ser más fácil implementar un semáforo binario, y puede demostrarse que tiene la misma potencia expresiva que un semáforo general (*véase* el Problema 5.9). Para contrastar los dos tipos de semáforos, el semáforo no-binario es a menudo referido como **semáforo con contador** o **semáforo general**.

Para ambos, semáforos con contador y semáforos binarios, se utiliza una cola para mantener los procesos esperando por el semáforo. Surge la cuestión sobre el orden en que los procesos deben ser

---

[3] En el artículo original de Dijkstra y en mucha de la literatura, se utiliza la letra `P` para `semWait` y la letra `V` para `semSignal`; estas son las iniciales de las palabras holandesas prueba (*probaren*) e incremento (*verhogen*). En alguna literatura se utilizan los términos `wait` y `signal`. Este libro utiliza `semWait` y `semSignal` por claridad y para evitar confusión con las operaciones similares *wait* y *signal* de los monitores, tratadas posteriormente.

```
struct semaphore {
    int cuenta;
    queueType cola;
}
void semWait(semaphore s)
{
    s.cuenta—;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso;
    }
}
void semSignal(semaphore s)
{
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
}
```

Figura 5.3. Una definición de las primitivas del semáforo.

```
struct binary_semaphore {
    enum {cero, uno} valor;
    queueType cola;
};
void semWaitB(binary_semaphore s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
        {
            poner este proceso en s.cola;
            bloquear este proceso;
    }
}
void semSignalB(binary_semaphore s)
{
    if (esta_vacia(s.cola))
        s.valor = 1;
    else
    {
            extraer un proceso P de s.cola;
            poner el proceso P en la lista de listos;
    }
}
```

Figura 5.4. Una definición de las primitivas del semáforo binario.

extraídos de tal cola. La política más favorable es FIFO (primero-en-entrar-primero-en-salir): el proceso que lleve más tiempo bloqueado es el primero en ser extraído de la cola; un semáforo cuya definición incluye esta política se denomina **semáforo fuerte**. Un semáforo que no especifica el orden en que los procesos son extraídos de la cola es un **semáforo débil**. La Figura 5.5, basada en otra de [DENN84], es un ejemplo de la operación de un semáforo fuerte. Aquí los procesos A, B y C dependen de un resultado del proceso D. Inicialmente (1), A está ejecutando; B, C y D están listos; y el contador del semáforo es 1, indicando que uno de los resultados de D está disponible. Cuando A realiza una instrucción `semWait` sobre el semáforo *s*, el semáforo se decrementa a 0 y A puede continuar ejecutando; posteriormente se adjunta a la lista de listos. Entonces B ejecuta (2), finalmente realiza una instrucción `semWait` y es bloqueado, permitiendo que D ejecute (3). Cuando D completa un nuevo resultado, realiza una instrucción `semSignal`, que permite a B moverse a la lista de listos (4). D se adjunta a la lista de listos y C comienza a ejecutar (5) pero se bloquea cuando realiza una instrucción `semWait`. De manera similar, A y B ejecutan y se bloquean en el semáforo, permitiendo a D retomar la ejecución (6). Cuando D tiene un resultado realiza un `semSignal`, que transfiere a C a la lista de listos. Posteriores ciclos de D liberarán A y B del estado Bloqueado.

Para el algoritmo de exclusión mutua tratado en la siguiente subsección e ilustrado en la Figura 5.6, los semáforos fuertes garantizan estar libres de inanición mientras que los semáforos débiles no. Se asumirán semáforos fuertes dado que son más convenientes y porque ésta es la forma típica del semáforo proporcionado por los sistemas operativos.

EXCLUSIÓN MUTUA

La Figura 5.6 muestra una solución directa al problema de la exclusión mutua usando un semáforo *s* (compárese con la Figura 5.1). Considere *n* procesos, identificados como *P(i)*, los cuales necesitan todos acceder al mismo recurso. Cada proceso tiene una sección crítica que accede al recurso. En cada proceso se ejecuta un semWait(*s*) justo antes de entrar en su sección crítica. Si el valor de *s* pasa a ser negativo, el proceso se bloquea. Si el valor es 1, entonces se decrementa a 0 y el proceso entra en su sección crítica inmediatamente; dado que *s* ya no es positivo, ningún otro proceso será capaz de entrar en su sección crítica.

El semáforo se inicializa a 1. Así, el primer proceso que ejecute un `semWait` será capaz de entrar en su sección crítica inmediatamente, poniendo el valor de *s* a 0. Cualquier otro proceso que intente entrar en su sección crítica la encontrará ocupada y se bloqueará, poniendo el valor de *s* a -1. Cualquier número de procesos puede intentar entrar, de forma que cada intento insatisfactorio conllevará otro decremento del valor de *s*. Cuando el proceso que inicialmente entró en su sección crítica salga de ella, *s* se incrementa y uno de los procesos bloqueados (si hay alguno) se extrae de la lista de procesos bloqueados asociada con el semáforo y se pone en estado Listo. Cuando sea planificado por el sistema operativo, podrá entrar en la sección crítica.

La Figura 5.7, basada en otra de [BACO03], muestra una posible secuencia de tres procesos usando la disciplina de exclusión mutua de la Figura 5.6. En este ejemplo, tres procesos (A, B, C) acceden a un recurso compartido protegido por el semáforo *s*. El proceso A ejecuta `semWait(s)`, y dado que el semáforo tiene el valor 1 en el momento de la operación `semWait`, A puede entrar inmediatamente en su sección crítica y el semáforo toma el valor 0. Mientras A está en su sección crítica, ambos B y C realizan una operación `semWait` y son bloqueados, pendientes de la disponibilidad del semáforo. Cuando A salga de su sección crítica y realice `semSignal(s)`, B, que fue el primer proceso en la cola, podrá entonces entrar en su sección crítica.

El programa de la Figura 5.6 puede servir igualmente si el requisito es que se permita más de un proceso en su sección crítica a la vez. Este requisito se cumple simplemente inicializando el semáforo al valor especificado. Así, en cualquier momento, el valor de *s.cuenta* puede ser interpretado como sigue:

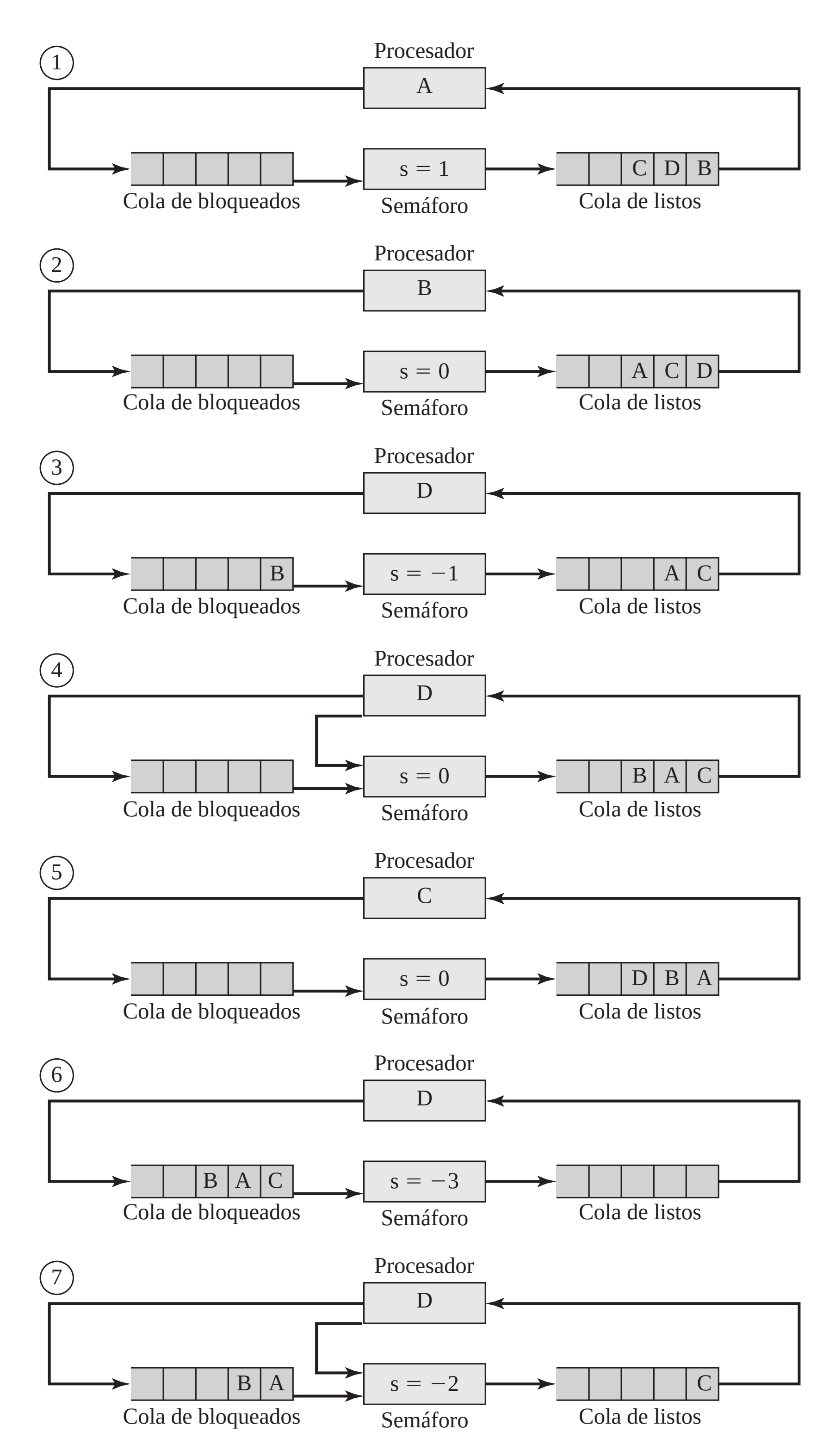


Figura 5.5. Ejemplo de mecanismo semáforo.

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
semafore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* sección crítica */;
          semSignal(s);
          /* resto */
     }
}
void main()
{
     paralelos (P(1), P(2), . . . ,P(n));
}
```

Figura 5.6. Exclusión mutua usando semáforos.

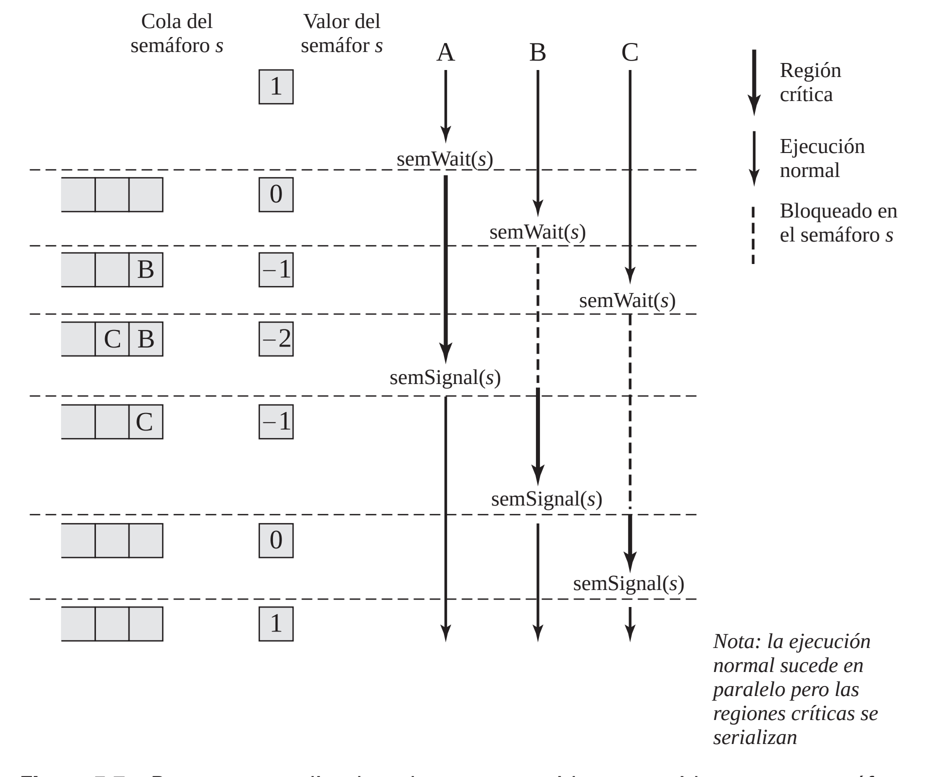


Figura 5.7. Procesos accediendo a datos compartidos protegidos por un semáforo.

- ***s.cuenta*** >= 0: *s.cuenta* es el número de procesos que pueden ejecutar `semWait(s)` sin suspensión (si no se ejecuta `semSignal(s)` entre medias). Tal situación permitirá a los semáforos admitir sincronización así como exclusión mutua.
- ***s.cuenta*** < 0: la magnitud de *s.cuenta* es el número de procesos suspendidos en *s.cola*.

EL PROBLEMA PRODUCTOR/CONSUMIDOR

Examinemos ahora uno de los problemas más comunes afrontados en programación concurrente: el problema productor/consumidor. El enunciado general es éste: hay uno o más procesos generando algún tipo de datos (registros, caracteres) y poniéndolos en un *buffer*. Hay un único consumidor que está extrayendo datos de dicho *buffer* de uno en uno. El sistema está obligado a impedir la superposición de las operaciones sobre los datos. Esto es, sólo un agente (productor o consumidor) puede acceder al *buffer* en un momento dado. Analizaremos varias soluciones a este problema para ilustrar tanto la potencia como las dificultades de los semáforos.

Para empezar, asúmase que el *buffer* es infinito y consiste en un vector de datos. En términos abstractos, las funciones `productor` y `consumidor` se definen como sigue:

```
productor:
while (true)
{
    /* producir dato v */;
    b[entra] = v;
    entra++;
}
```

```
consumidor:
while (true)
{
    while (entra <= sale)
        /* no hacer nada */;
    w = b[sale];
    sale++;
    /* consumir dato w */
}
```

La Figura 5.8 ilustra la estructura del *buffer* `b.` El productor puede generar datos y almacenarlos en el vector a su propio ritmo. Cada vez, se incrementa un índice (*entra*) sobre el vector. El consumidor procede de manera similar pero debe asegurarse de que no intenta leer de una entrada vacía del vector. Por tanto, el consumidor se asegura de que el productor ha avanzado más allá que él (*entra* > *sale*) antes de seguir.

Intentemos implementar este sistema utilizando semáforos binarios. La Figura 5.9 es un primer intento. En vez de tratar con los índices *entra* y *sale*, simplemente guardamos constancia del número

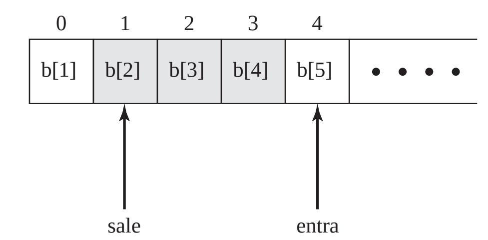


Figura 5.8. *Buffer* infinito para el problema productor/consumidor.

de datos en el *buffer* usando la variable entera *n* (= *entra* – *sale*). El semáforo *s* se utiliza para cumplir la exclusión mutua; el semáforo *retardo* se usa para forzar al consumidor a esperar (`semWait`) si el *buffer* está vacío.

```
/* programa productor consumidor */
int n;
binary_semaphore s = 1;
binary_semaphore retardo = 0;
void productor()
{
	while (true)
	{
			producir();
			semWaitB(s);
			anyadir();
			n++;
			if (n==1)
					semSignalB(retardo);
			semSignalB(s);
	}
}
void consumidor()
{
	semWaitB(retardo);
	while (true)
	{
			semWaitB(s);
			extraer();
			n—;
			semSignalB(s);
			consumir();
			if (n==0)
					semWaitB(retardo);
	}
}
void main()
{
	n = 0;
	paralelos (productor, consumidor);
}
```

Figura 5.9. Una solución incorrecta al problema productor/consumidor con *buffer* infinito usando semáforos binarios.

Esta solución parece bastante directa. El productor es libre de añadir al *buffer* en cualquier momento. Realiza `semWaitB(s)` antes de añadir y `semSignalB(s)` tras haberlo hecho, para impedir que el consumidor, o cualquier otro productor, acceda al *buffer* durante la operación de añadir el dato. También, mientras está en su sección crítica, el productor incrementa el valor de *n*. Si *n* = 1, entonces el *buffer* estaba vacío justo antes de este añadido, por lo que el productor realiza un `semSignalB(retardo)` para alertar al consumidor de este hecho. El consumidor comienza esperando a que se produzca el primer dato, usando `semWaitB(retardo)`. Luego toma el dato y decrementa *n* en su sección crítica. Si el productor es capaz de permanecer por delante del consumidor (una situación usual), entonces el consumidor raramente se bloqueará en el semáforo `retardo` porque `n` será normalmente positivo. Por tanto, productor y consumidor avanzarán ambos sin problemas.

Sin embargo, hay un defecto en este programa. Cuando el consumidor ha agotado los datos del *buffer*, necesita restablecer el semáforo `retardo` para obligarse a esperar hasta que el productor haya introducido más datos en el *buffer*. Este el propósito de la sentencia: `if (n == 0) semWaitB(retardo)`. Considere el escenario esbozado en la Tabla 5.3. En la línea 14, el consumidor no ejecuta la operación `semWaitB`. El consumidor, efectivamente, ha vaciado el *buffer* y ha puesto *n* a 0 (línea 8), pero el productor ha incrementado *n* antes de que el consumidor pueda comprobar su valor en la línea 14. El resultado es un `semSignalB` que no casa con un `semWaitB` anterior. El valor de -1 en *n* en la línea 20 significa que el consumidor ha consumido del *buffer* un dato inexistente. No cabe simplemente mover la sentencia condicional dentro de la sección crítica del consumidor, porque esto podría dar lugar a un interbloqueo (por ejemplo, tras la línea 8 de la tabla).

Una solución para este problema es introducir una variable auxiliar que pueda establecerse dentro de la sección crítica del consumidor para su uso posterior. Esto se muestra en la Figura 5.10. El análisis cuidadoso de la lógica del código debería convencer de que el interbloqueo ya no puede suceder.

Si se utilizan semáforos generales (también denominados semáforos con contador) puede obtenerse una solución algo más clara, como se muestra en la Figura 5.11. La variable `n` es ahora un semáforo. Su valor sigue siendo en número de datos en el *buffer*. Supóngase ahora que en la transcripción de este programa, se comete un error y se intercambian las operaciones `semSignal(s)` y `semSignal(n)`. Esto requeriría que la operación `semSignal(n)` se realizase en la sección crítica del productor sin interrupción por parte del consumidor o de otro productor. ¿Afectaría esto al programa? No, porque el consumidor debe, en cualquier caso, esperar por ambos semáforos antes de seguir.

Supóngase ahora que las operaciones `semWait(n)` y `semWait(s)` se intercambian accidentalmente. Esto produce un serio error, de hecho fatal. Si el consumidor entrase alguna vez en su sección crítica cuando el *buffer* está vacío (*n.cuenta* = 0), entonces ningún productor podrá nunca añadir al *buffer* y el sistema estará interbloqueado. Éste es un buen ejemplo de la sutileza de los semáforos y de la dificultad de producir diseños correctos.

Finalmente, añadamos una restricción nueva y realista al problema productor/consumidor: a saber, que el *buffer* es finito. El *buffer* se trata como un *buffer* circular (Figura 5.12), y los valores que lo indexan deben ser expresados módulo el tamaño del *buffer*. Se mantiene la siguiente relación:

Bloquearse cuando:	**Desbloquear cuando:**
Productor: al insertar con el *buffer* lleno	Consumidor: dato insertado
Consumidor: al extraer con el *buffer* vacío	Productor: dato extraído

Tabla 5.3. Posible Escenario para el Programa de la Figura 5.9.

	Productor	Consumidor	s	n	Retardo
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) semSignalB(retardo)		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(retardo)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) semSignalB(retardo)		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) semWaitB(retardo)	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) semWaitB(retardo)	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

Las áreas en blanco representan la sección crítica controlada por semáforos.

Las funciones **productor** y **consumidor** pueden expresarse como sigue (las variables *entra* y *sale* están inicializadas a 0):

```
productor:
while (true)
{
    /* producir dato v */;
    while ((entra + 1) % n == sale)
        /* no hacer nada */;
    b[entra] = v;
    entra = (entra + 1) % n;
}
```

```
consumidor:
while (true)
{
    while (entra == sale)
    /* no hacer nada */;
    w = b[sale];
    sale = (sale + 1) % n;
    /* consumir dato w */
}
```

```
/* programa productor consumidor */
int n;
binary_semaphore s = 1;
binary_semaphore retardo = 0;
void productor()
{
      while (true)
      {
              producir();
              semWaitB(s);
              anyadir();
              n++;
              if (n==1) semSignalB(retardo);
              semSignalB(s);
      }
}
void consumidor()
{
      int m; /* una variable local */
      semWaitB(retardo);
      while (true)
      {
              semWaitB(s);
              extraer();
              n—;
              m = n;
              semSignalB(s);
              consumir();
              if (m==0) semWaitB(retardo);
      }
}
void main()
{
      n = 0;
      paralelos (productor, consumidor);
}
```

Figura 5.10. Una solución correcta al problema productor/consumidor con *buffer* infinito usando semáforos binarios.

```
/* programa productor consumidor */
semaphore n = 0;
semaphore s = 1;
void productor()
{
	while (true)
	{
			producir();
			semWait(s);
			anyadir();
			semSignal(s);
			semSignal(n);
	}
}
void consumidor()
{
	while (true)
	{
			semWait(n);
			semWait(s);
			extraer();
			semSignal(s);
			consumir();
	}
}
void main()
{
	paralelos (productor, consumidor);
}
```

Figura 5.11. Una solución al problema productor/consumidor con *buffer* infinito usando semáforos.

La Figura 5.13 muestra una solución usando semáforos generales. El semáforo *e* se ha añadido para llevar la cuenta del número de espacios vacíos.

Otro instructivo ejemplo del uso de los semáforos es el problema de la barbería, descrito en el Apéndice A. El Apéndice A también incluye ejemplos adicionales del problema de las condiciones de carrera cuando se usan semáforos.

IMPLEMENTACIÓN DE SEMÁFOROS

Como se mencionó anteriormente, es imperativo que las funciones `semWait` y `semSignal` sean implementadas como primitivas atómicas. Una manera obvia es implementarlas en hardware o *firmware*. A falta de esto, se han propuesto variedad de esquemas. La esencia del problema es la

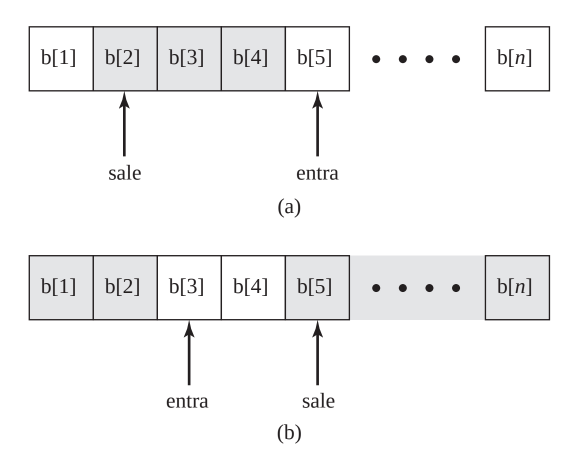


Figura 5.12. *Buffer* finito circular para el problema productor/consumidor.

```
/* programa productor consumidor */
semaphore s = 1;
semaphore n = 0;
semaphore e = /* tamaño del buffer */;
void productor()
{
      while (true)
      {
                  producir();
                  semWait(e);
                  semWait(s);
                  anyadir();
                  semSignal(s);
                  semSignal(n);
      }
}
void consumidor()
{
      while (true)
      {
                  semWait(n);
                  semWait(s);
                  extraer();
                  semSignal(s);
                  semSignal(e);
                  consumir();
      }
}
void main()
{
      paralelos (productor, consumidor);
}
```

Figura 5.13. Una solución al problema productor/consumidor con *buffer* acotado usando semáforos.

de la exclusión mutua: sólo un proceso al tiempo puede manipular un semáforo bien con la operación `semWait` o bien con `semSignal`. Así, cualquiera de los esquemas software, tales como el algoritmo de Dekker o el de Peterson (Apéndice A), pueden usarse, si bien esto supondría una substancial sobrecarga de procesamiento. Otra alternativa es utilizar uno de los esquemas soportados por hardware para la exclusión mutua. Por ejemplo, la Figura 5.14a muestra la utilización de la instrucción *test and set*. En esta implementación, el semáforo es nuevamente una estructura como en la Figura 5.3, pero ahora incluye un nuevo componente entero, *s.ocupado*. Lo cierto es que involucra cierta forma de espera activa. Sin embargo, las operaciones `semWait` y `semSignal` son relativamente cortas, luego la cantidad de espera activa implicada no debería ser relevante.

En un sistema de procesador único, es posible inhibir interrupciones durante las operaciones `semWait` y `semSignal`, tal y como se sugiere en la Figura 5.14b. Una vez más, la relativamente corta duración de estas operaciones significa que esta solución es razonable.

```
semWait(s)
{
    while (!testset(s.ocupado))
        /* no hacer nada */;
    s.cuenta––;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso
        (además poner s.ocupado a 0);
    }
    else
        s.ocupado = 0;
}
semSignal(s)
{
    while (!testset(s.ocupado))
        /* no hacer nada */;
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
    s.ocupado = 0;
}
```

(a) Instrucción `Testset`

```
semWait(s)
{
    inhibir interrupciones;
    s.cuenta––;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso
        y habilitar interrupciones;
    }
    else
        habilitar interrupciones;
}
semSignal(s)
{
    inhibir interrupciones;
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
    habilitar interrupciones;
}
```

(b) Interrupciones

Figura 5.14. Dos posibles implementaciones de semáforos.

5.4. MONITORES

Los semáforos proporcionan una herramienta potente y flexible para conseguir la exclusión mutua y para la coordinación de procesos. Sin embargo, como la Figura 5.9 sugiere, puede ser difícil producir un programa correcto utilizando semáforos. La dificultad es que las operaciones `semWait` y `semSignal` pueden estar dispersas a través de un programa y no resulta fácil ver el efecto global de estas operaciones sobre los semáforos a los que afectan.

El monitor es una construcción del lenguaje de programación que proporciona una funcionalidad equivalente a la de los semáforos pero es más fácil de controlar. El concepto se definió formalmente por primera vez en [HOAR74]. La construcción monitor ha sido implementada en cierto número de lenguajes de programación, incluyendo Pascal Concurrente, Pascal-Plus, Modula-2, Modula-3 y Java. También ha sido implementada como una biblioteca de programa. Esto permite a los programadores poner cerrojos monitor sobre cualquier objeto. En concreto, para algo como una lista encadenada, puede quererse tener un único cerrojo para todas las listas, por cada lista o por cada elemento de cada lista.

Comencemos viendo la versión de Hoare para luego examinar otra más refinada.

MONITOR CON SEÑAL

Un monitor es un módulo software consistente en uno o más procedimientos, una secuencia de inicialización y datos locales. Las principales características de un monitor son las siguientes:

1. Las variables locales de datos son sólo accesibles por los procedimientos del monitor y no por ningún procedimiento externo.
2. Un proceso entra en el monitor invocando uno de sus procedimientos.
3. Sólo un proceso puede estar ejecutando dentro del monitor al tiempo; cualquier otro proceso que haya invocado al monitor se bloquea, en espera de que el monitor quede disponible.

Las dos primeras características guardan semejanza con las de los objetos en el software orientado a objetos. De hecho, en un sistema operativo o lenguaje de programación orientado a objetos puede implementarse inmediatamente un monitor como un objeto con características especiales.

Al cumplir la disciplina de sólo un proceso al mismo tiempo, el monitor es capaz de proporcionar exclusión mutua fácilmente. Las variables de datos en el monitor sólo pueden ser accedidas por un proceso a la vez. Así, una estructura de datos compartida puede ser protegida colocándola dentro de un monitor. Si los datos en el monitor representan cierto recurso, entonces el monitor proporciona la función de exclusión mutua en el acceso al recurso.

Para ser útil para la programación concurrente, el monitor debe incluir herramientas de sincronización. Por ejemplo, suponga un proceso que invoca a un monitor y mientras está en él, deba bloquearse hasta que se satisfaga cierta condición. Se precisa una funcionalidad mediante la cual el proceso no sólo se bloquee, sino que libere el monitor para que algún otro proceso pueda entrar en él. Más tarde, cuando la condición se haya satisfecho y el monitor esté disponible nuevamente, el proceso debe poder ser retomado y permitida su entrada en el monitor en el mismo punto en que se suspendió.

Un monitor soporta la sincronización mediante el uso de **variables condición** que están contenidas dentro del monitor y son accesibles sólo desde el monitor. Las variables condición son un tipo de datos especial en los monitores que se manipula mediante dos funciones:

- `cwait(c)`: Suspende la ejecución del proceso llamante en la condición *c*. El monitor queda disponible para ser usado por otro proceso.

- `csignal(c)`: Retoma la ejecución de algún proceso bloqueado por un **cwait** en la misma condición. Si hay varios procesos, elige uno de ellos; si no hay ninguno, no hace nada.

Nótese que las operaciones *wait* y *signal* de los monitores son diferentes de las de los semáforos. Si un proceso en un monitor *señala* y no hay ningún proceso esperando en la variable condición, la señal se pierde.

La Figura 5.15 ilustra la estructura de un monitor. Aunque un proceso puede entrar en el monitor invocando cualquiera de sus procedimientos, puede entenderse que el monitor tiene un único punto de entrada que es el protegido, de ahí que sólo un proceso pueda estar en el monitor a la vez. Otros procesos que intenten entrar en el monitor se unirán a una cola de procesos bloqueados esperando por la disponibilidad del monitor. Una vez que un proceso está en el monitor, puede temporalmente bloquearse a sí mismo en la condición *x* realizando un `cwait(x);` en tal caso, el proceso será añadido a una cola de procesos esperando a reentrar en el monitor cuando cambie la condición y retomar la ejecución en el punto del programa que sigue a la llamada `cwait(x)`.

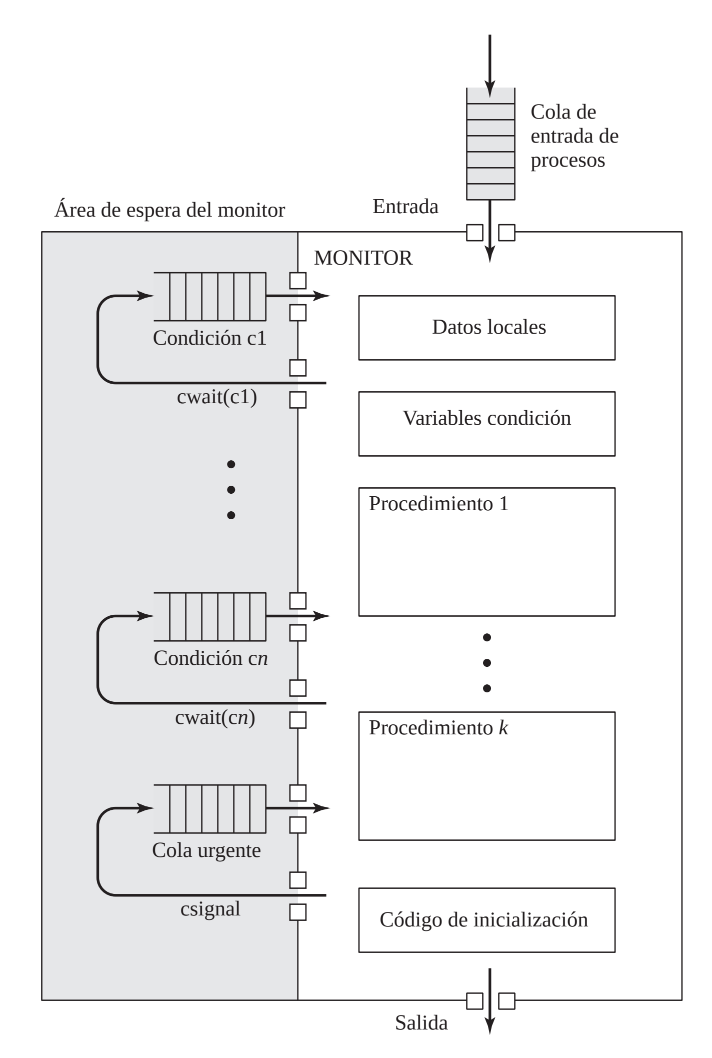


Figura 5.15. Estructura de un monitor.

Si un proceso que está ejecutando en el monitor detecta un cambio en la variable condición `x`, realiza un `csignal(x)`, que avisa del cambio a la correspondiente cola de la condición.

Como un ejemplo del uso de un monitor, retornemos al problema productor/consumidor con *buffer* acotado. La Figura 5.16 muestra una solución utilizando monitores. El módulo monitor, `buffer-acotado`, controla el *buffer* utilizado para almacenar y extraer caracteres. El monitor incluye dos variables condición (declaradas con la construcción `cond`): *nolleno* es cierta cuando hay espacio para añadir al menos un carácter al *buffer* y *novacio* es cierta cuando hay al menos un carácter en el *buffer*.

Un productor puede añadir caracteres al *buffer* sólo por medio del procedimiento `anyadir` dentro del monitor; el productor no tiene acceso directo a *buffer*. El procedimiento comprueba primero la condición *nolleno* para determinar si hay espacio disponible en el *buffer*. Si no, el proceso que ejecuta el monitor se bloquea en esa condición. Algún otro proceso (productor o consumidor) puede entrar ahora en el monitor. Más tarde, cuando el *buffer* ya no esté lleno, el proceso bloqueado podrá ser extraído de la cola, reactivado y retomará su labor. Tras colocar el carácter en el *buffer*, el proceso señala la condición *novacio*. Una descripción similar puede realizarse de la función del consumidor.

Este ejemplo marca la división de responsabilidad en los monitores en comparación con los semáforos. En el caso de los monitores, la construcción del monitor impone en sí misma la exclusión mutua: no es posible que ambos, productor y consumidor, accedan simultáneamente al *buffer*. Sin embargo, el programador debe disponer las primitivas `cwait` y `csignal` apropiadas en el código del monitor para impedir que se depositen datos cuando el *buffer* está lleno o que se extraigan cuando está vacío. En el caso de los semáforos, tanto la exclusión mutua como la sincronización son responsabilidad del programador.

Nótese que en la Figura 5.16 un proceso sale del monitor inmediatamente después de ejecutar la función `csignal`. Si `csignal` no sucede al final del procedimiento, entonces, en la propuesta de Hoare, el proceso que emite la señal se bloquea para que el monitor pase a estar disponible, y se sitúa en una cola hasta que el monitor sea liberado. En este punto, una posibilidad sería colocar el proceso bloqueado en la cola de entrada, de manera que tendría que competir por el acceso con otros procesos que no han entrado todavía en el monitor. No obstante, dado que un proceso bloqueado en una función `csignal` ha realizado ya parcialmente su tarea en el monitor, tiene sentido dar preferencia a este proceso sobre los procesos que entraron recientemente, disponiéndolo en una cola urgente separada (Figura 5.15). Uno de los lenguajes que utiliza monitores, Pascal Concurrente, exige que `csignal` aparezca solamente como última operación ejecutada por un procedimiento monitor.

Si no hay procesos esperando en la condición *x*, entonces la ejecución de `csignal(x)` no tiene efecto.

Como con los semáforos, es posible cometer errores en la función de sincronización de los monitores. Por ejemplo, si se omite alguna de las funciones `csignal` en el monitor `bufferacotado`, entonces los procesos que entren en la cola de la correspondiente condición estarán permanentemente colgados. La ventaja que los monitores tienen sobre los semáforos es que todas las funciones de sincronización están confinadas en el monitor. Por tanto, es más fácil comprobar que la sincronización se ha realizado correctamente y detectar los errores. Es más, una vez un monitor se ha programado correctamente, el acceso al recurso protegido será correcto para todo acceso desde cualquier proceso. En cambio, con los semáforos, el acceso al recurso será correcto sólo si todos los procesos que acceden al recurso han sido programados correctamente.

MODELO ALTERNATIVO DE MONITORES CON NOTIFICACIÓN Y DIFUSIÓN

La definición de monitor de Hoare [HOAR74] requiere que si hay al menos un proceso en una cola de una condición, un proceso de dicha cola ejecuta inmediatamente cuando otro proceso realice un

```
/* programa productor consumidor */
monitor bufferacotado;
char buffer[N];                                              /* espacio para N datos */
int dentro, fuera;                                           /* punteros al buffer */
int cuenta;                                                  /* número de datos en el buffer */
cond nolleno, novacio;                                       /* variables condición para sincronizar */
void anyadir (char x)
{
    if (cuenta == N)
        cwait(nolleno);                                      /* buffer lleno, evitar desbordar */
    buffer[dentro] = x;
    dentro = (dentro + 1) % N;
    cuenta++;                                                /* un dato más en el buffer */
    csignal(novacio);                                        /* retoma algún consumidor en espera */
}
void extraer (char x)
{
    if (cuenta == 0)
        cwait(novacio);                                      /* buffer vacío, evitar consumo */
    x = buffer[fuera];
    fuera = (fuera + 1) % N;
    cuenta—;                                                 /* un dato menos en el buffer */
    csignal(nolleno);                                        /* retoma algún productor en espera */
}
{                                                            /* cuerpo del monitor */
    dentro = 0; fuera = 0; cuenta = 0;                       /* buffer inicialmente vacío */
}
```

```
void productor()
{
    char x;
    while (true)
    {
        producir(x);
        anyadir(x);
    }
}
void consumidor()
{
    char x;
    while (true)
    {
        extraer(x);
        consumir(x);
    }
}
void main()
{
    paralelos (productor, consumidor);
}
```

Figura 5.16. Una solución al problema productor/consumidor con *buffer* acotado usando un monitor.

`csignal` sobre dicha condición. Así, el proceso que realiza el `csignal` debe bien salir inmediatamente del monitor o bien bloquearse dentro del monitor.

Hay dos desventajas en esta solución:

1. Si el proceso que realiza el `csignal` no ha terminado con el monitor, entonces se necesitarán dos cambios de proceso adicionales: uno para bloquear este proceso y otro para retomarlo cuando el monitor quede disponible.
2. La planificación de procesos asociada con una señal debe ser perfectamente fiable. Cuando se realiza un `csignal,` un proceso de la cola de la correspondiente condición debe ser activado inmediatamente y el planificador debe asegurar que ningún otro proceso entra en el monitor antes de la activación. De otro modo, la condición bajo la cual el proceso fue activado podría cambiar. Por ejemplo, en la Figura 5.16, cuando se realiza un `csignal(`*`novacio`*`),` debe ser activado un proceso de la cola *`novacio`* antes de que un nuevo consumidor entre en el monitor. Otro ejemplo: un proceso productor puede añadir un carácter a un *buffer* vacío y entonces fallar justo antes de la señalización; los procesos de la cola *`novacio`* estarían permanentemente colgados.

Lampson y Redell desarrollaron una definición diferente de monitor para el lenguaje Mesa [LAMP80]. Su solución resuelve los problemas que se acaban de enunciar y aporta varias extensiones útiles. La estructura del monitor de Mesa se utiliza también en el lenguaje de programación de sistemas Modula-3 [NELS91]. En Mesa, la primitiva `csignal` se sustituye por la `cnotify,` con la siguiente interpretación: cuando un proceso ejecutando un monitor ejecuta `cnotify(`*`x`*`),` provoca que la cola de la condición *x* sea notificada, pero el proceso que señaló continúa ejecutando. El resultado de la notificación es que el proceso en cabeza de la cola de la condición será retomado en un momento futuro conveniente, cuando el monitor esté disponible. Sin embargo, como no hay garantía de que algún otro proceso entre en el monitor antes que el proceso notificado, el proceso notificado deberá volver a comprobar la condición. Por ejemplo, los procedimientos en el monitor *`buffer`*`acotado` tendrían ahora el código de la Figura 5.17.

```
void anyadir (char x)
{
    while (cuenta == N)
        cwait(nolleno);                         /* buffer lleno, evitar desbordar */
    buffer[dentro] = x;
    dentro = (dentro + 1) % N;
    cuenta++;                                   /* un dato más en el buffer */
    cnofify(novacio);                           /* notifica a algún consumidor en espera */
}
void extraer (char x)
{
    while (cuenta == 0)
        cwait(novacio);                         /* buffer vacío, evitar consumo */
    x = buffer[fuera];
    fuera = (fuera + 1) % N;
    cuenta—;                                    /* un dato menos en el buffer */
    cnotify(nolleno);                           /* notifica a algún productor en espera */
}
```

Figura 5.17. Monitor de *buffer* acotado con el monitor Mesa.

Las sentencias **if** se remplazan por bucles **while**. Así, este convenio requiere al menos una evaluación extra de la variable condición. A cambio, sin embargo, no hay cambios de proceso extra ni tampoco hay restricciones sobre cuando, tras el `cnotify,` debe ejecutar el proceso notificado.

Una mejora útil que puede asociarse con la primitiva `cnotify` es un temporizador asociado con cada primitiva de condición. Un proceso que haya estado esperando el máximo del intervalo de tiempo indicado, será situado en estado Listo con independencia de que la condición haya sido notificada. Cuando sea activado, el proceso comprobará la condición y continuará si la condición se satisfizo. La temporización evita la inanición indefinida de un proceso en el caso de que algún otro proceso falle antes de señalar la condición.

Con la norma de que un proceso se notifica en vez de que se reactiva por la fuerza, es posible añadir la primitiva `cbroadcast` al repertorio. La difusión (*broadcast*) provoca que todos los procesos esperando en una condición pasen a estado Listo. Esto es conveniente en situaciones donde un proceso no sabe cuántos otros procesos deben ser reactivados. Por ejemplo, en el programa productor/consumidor, suponga que ambas funciones `anyadir` y `extraer` puedan aplicarse a bloques de caracteres de longitud variable. En este caso, si un productor añade un bloque de caracteres al *buffer*, no necesita saber cuántos caracteres está dispuesto a consumir cada consumidor en espera. Simplemente emite un `cbroadcast` y todos los procesos en espera serán avisados para que lo intenten de nuevo.

En suma, puede usarse un `cbroadcast` cuando un proceso tenga dificultad en conocer de manera precisa cuántos otros procesos debe reactivar. Un buen ejemplo es un gestor de memoria. El gestor tiene j bytes libres; un proceso libera k bytes adicionales, pero no se sabe si algún proceso en espera puede seguir con un total de $k + j$ bytes. Por tanto utiliza la difusión y todos los procesos verifican por sí mismos si hay suficiente memoria libre.

Una ventaja de los monitores de Lampson/Redell sobre los monitores de Hoare es que la solución de Lampson/Redell es menos propensa a error. En la solución de Lampson/Redell, dado que, al usarse la construcción **while**, cada procedimiento comprueba la variable condición después de ser señalado, un proceso puede señalar o difundir incorrectamente sin causar un error en el programa señalado. El programa señalado comprobará la variable relevante y si la condición no se cumple, volverá a esperar.

Otra ventaja del monitor Lampson/Redell es que se presta a un enfoque más modular de la construcción de programas. Por ejemplo, considere la implementación de la reserva de un *buffer* de E/S. Hay dos niveles de condiciones que deben ser satisfechas por los procesos secuenciales cooperantes:

1. Estructuras de datos concordantes. Así, el monitor cumple la exclusión mutua y completa una operación de entrada o salida antes de permitir otra operación sobre el *buffer*.

2. Nivel 1, más suficiente memoria para que este proceso pueda completar su solicitud de reserva.

En el monitor de Hoare, cada señal transporta la condición de nivel 1 pero también lleva un mensaje implícito, «He liberado suficientes bytes para que tu llamada de solicitud de reserva pueda ahora funcionar». Así, la señal lleva implícita la condición de nivel 2. Si el programador cambia más tarde la definición de la condición de nivel 2, será necesario reprogramar todos los procesos que señalizan. Si el programador cambia las suposiciones realizadas por cualquier proceso en espera concreto (esto es, esperar por un invariante ligeramente diferente al nivel 2), podría ser necesario reprogramar todos los procesos que señalizan. Esto no es modular y es propenso a causar errores de sincronización (por ejemplo, despertar por error) cuando se modifica el código. El programador ha de acordarse de modificar todos los procedimientos del monitor cada vez que se realiza un pequeño cambio en la condición de nivel 2. Con un monitor Lampson/Redell, una difusión asegura la condición de nivel 1 e insinúa que puede que se cumpla la de nivel 2; cada proceso debe comprobar la condición de nivel 2 por

sí mismo. Si se realiza un cambio en la condición de nivel 2, bien en quién espera bien en quién señala, no hay posibilidad de un despertar erróneo porque cada procedimiento verifica su propia condición de nivel 2. Por tanto, la condición de nivel 2 puede quedar oculta dentro de cada procedimiento. Con el monitor de Hoare, la condición de nivel 2 debe trasladarse desde el código del proceso en espera hasta el código de cada uno de los procesos que señalizan, lo cual viola la abstracción de datos y los principios de la modularidad interprocedural.

5.5. PASO DE MENSAJES

Cuando los procesos interaccionan entre sí, deben satisfacerse dos requisitos fundamentales: sincronización y comunicación. Los procesos necesitan ser sincronizados para conseguir exclusión mutua; los procesos cooperantes pueden necesitar intercambiar información. Un enfoque que proporciona ambas funciones es el paso de mensajes. El paso de mensajes tiene la ventaja añadida de que se presta a ser implementado tanto en sistemas distribuidos como en multiprocesadores de memoria compartida y sistemas monoprocesador.

Los sistemas de paso de mensajes se presentan en varias modalidades. En esta sección, presentamos una introducción general que trata las características típicas encontradas en tales sistemas. La funcionalidad real del paso de mensajes se proporciona normalmente en forma de un par de primitivas:

send(destino, mensaje)

receive(origen, mensaje)

Este es el conjunto mínimo de operaciones necesarias para que los procesos puedan entablar paso de mensajes. El proceso envía información en forma de un *mensaje* a otro proceso designado por *destino*. El proceso recibe información ejecutando la primitiva **receive**, indicando la *fuente* y el *mensaje*.

Tabla 5.4. Características de diseño en sistemas de mensajes para comunicación y sincronización interprocesador.

Sincronización	**Formato**
Send	Contenido
Bloqueante	Longitud
No bloqueante	Fija
Receive	Variable
Bloqueante	
No bloqueante	
Comprobación de llegada	
Direccionamiento	**Disciplina de cola**
Directo	FIFO
Send	Prioridad
Receive	
Explícito	
Implícito	
Indirecto	
Estático	
Dinámico	
Propiedad	

En la Tabla 5.4 se muestran cierto número de decisiones de diseño relativas a los sistemas de paso de mensaje que van a ser examinadas en el resto de esta sección.

SINCRONIZACIÓN

La comunicación de un mensaje entre dos procesos implica cierto nivel de sincronización entre los dos: el receptor no puede recibir un mensaje hasta que no lo haya enviado otro proceso. En suma, tenemos que especificar qué le sucede a un proceso después de haber realizado una primitiva `send` o `receive`.

Considérese primero la primitiva `send.` Cuando una primitiva `send` se ejecuta en un proceso, hay dos posibilidades: o el proceso que envía se bloquea hasta que el mensaje se recibe o no se bloquea. De igual modo, cuando un proceso realiza la primitiva `receive,` hay dos posibilidades:

1. Si el mensaje fue enviado previamente, el mensaje será recibido y la ejecución continúa.
2. Si no hay mensajes esperando, entonces: (a) el proceso se bloquea hasta que el mensaje llega o (b) el proceso continúa ejecutando, abandonando el intento de recepción.

Así, ambos emisor y receptor pueden ser bloqueantes o no bloqueantes. Tres son las combinaciones típicas, si bien un sistema en concreto puede normalmente implementar sólo una o dos de las combinaciones:

- **Envío bloqueante, recepción bloqueante.** Ambos emisor y receptor se bloquean hasta que el mensaje se entrega; a esto también se le conoce normalmente como *rendezvous*.
- **Envío no bloqueante, recepción bloqueante.** Aunque el emisor puede continuar, el receptor se bloqueará hasta que el mensaje solicitado llegue. Esta es probablemente la combinación más útil.
- **Envío no bloqueante, recepción no bloqueante.** Ninguna de las partes tiene que esperar.

Para muchas tareas de programación concurrente es más natural el `send` no bloqueante. Por ejemplo, si se utiliza para realizar una operación de salida, como imprimir, permite que el proceso solicitante emita la petición en forma de un mensaje y luego continúe. Un peligro potencial del `send` no bloqueante es que un error puede provocar una situación en la cual los procesos generan mensajes repetidamente. Dado que no hay bloqueo que castigue al proceso, los mensajes podrían consumir recursos del sistema, incluyendo tiempo de procesador y espacio de almacenamiento, en detrimento de otros procesos y del sistema operativo. También, el envío no bloqueante pone sobre el programador la carga de determinar si un mensaje ha sido recibido: los procesos deben emplear mensajes de respuesta para reconocer la recepción de un mensaje.

Para la primitiva `receive,` la versión bloqueante parece ser la más natural para muchas tareas de programación concurrente. Generalmente, un proceso que quiere un mensaje necesita esperar la información antes de continuar. No obstante, si un mensaje se pierde, lo cual puede suceder en un sistema distribuido, o si un proceso falla antes de enviar un mensaje que se espera, el proceso receptor puede quedar bloqueado indefinidamente. Este problema puede resolverse utilizando el `receive` no bloqueante. Sin embargo, el peligro de este enfoque es que si un mensaje se envía después de que un proceso haya realizado el correspondiente `receive,` el mensaje puede perderse. Otras posibles soluciones son permitir que el proceso receptor compruebe si hay un mensaje en espera antes de realizar el `receive` y permitirle al proceso especificar más de un origen en la primitiva `receive.` La segunda solución es útil si un proceso espera mensajes de más de un posible origen y puede continuar si llega cualquiera de esos mensajes.

DIRECCIONAMIENTO

Claramente, es necesario tener una manera de especificar en la primitiva de envío qué procesos deben recibir el mensaje. De igual modo, la mayor parte de las implementaciones permiten al proceso receptor indicar el origen del mensaje a recibir.

Los diferentes esquemas para especificar procesos en las primitivas `send` y `receive` caben dentro de dos categorías: direccionamiento directo y direccionamiento indirecto. Con el **direccionamiento directo**, la primitiva `send` incluye un identificador específico del proceso destinatario. La primitiva `receive` puede ser manipulada de dos maneras. Una posibilidad es que el proceso deba designar explícitamente un proceso emisor. Así, el proceso debe conocer con anticipación de qué proceso espera el mensaje. Esto suele ser lo más eficaz para procesos concurrentes cooperantes. En otros casos, sin embargo, es imposible especificar con anticipación el proceso de origen. Un ejemplo es un proceso servidor de impresión, que deberá aceptar un mensaje de solicitud de impresión de cualquier otro proceso. Para tales aplicaciones, una solución más efectiva es el uso de direccionamiento implícito. En este caso, el parámetro *origen* de la primitiva `receive` toma un valor devuelto por la operación de recepción cuando se completa.

El otro esquema general es el **direccionamiento indirecto**. En este caso, los mensajes no se envían directamente por un emisor a un receptor sino que son enviados a una estructura de datos compartida que consiste en colas que pueden contener mensajes temporalmente. Tales colas se conocen generalmente como buzones (*mailboxes*). Así, para que dos procesos se comuniquen, un proceso envía un mensaje al buzón apropiado y otro proceso toma el mensaje del buzón.

Una virtud del uso del direccionamiento indirecto es que, desacoplando emisor y receptor, se permite una mayor flexibilidad en el uso de mensajes. La relación entre emisores y receptores puede ser uno-a-uno, muchos-a-uno, uno-a-muchos o muchos-a-muchos (Figura 5.18). Una relación **uno-a-uno** permite establecer un enlace de comunicaciones privadas entre dos procesos. Esto aísla su interacción de interferencias erróneas de otros procesos. Una relación **muchos-a-uno** es útil para interacciones cliente/servidor; un proceso proporciona servicio a otros muchos procesos. En este caso, el buzón se conoce normalmente como *puerto*. Una relación **uno-a-muchos** permite un emisor y múltiples receptores; esto es útil para aplicaciones donde un mensaje, o cierta información, debe ser difundido a un conjunto de procesos. Una relación **muchos-a-muchos** permite a múltiples procesos servidores proporcionar servicio concurrente a múltiples clientes.

La asociación de procesos a buzones puede ser estática o dinámica. Normalmente los puertos se asocian estáticamente con un proceso en particular; esto es, el puerto se crea y asigna a un proceso permanentemente. De igual modo, normalmente una relación **uno-a-uno** se define estática y permanentemente. Cuando hay varios emisores, la asociación de un emisor a un buzón puede ocurrir dinámicamente. Para este propósito pueden utilizarse primitivas como `connect` y `disconnect`.

Un aspecto relacionado tiene que ver con la propiedad del buzón. En el caso de un puerto, típicamente es creado por (y propiedad de) el proceso receptor. Así, cuando se destruye el proceso, el puerto también. Para el caso general de buzón, el sistema operativo puede ofrecer un servicio para crearlos. Tales buzones pueden verse bien como propiedad del proceso que lo crea, en cuyo caso se destruye junto con el proceso, o bien propiedad del sistema operativo, en cuyo caso se precisa un mandato explícito para destruir el buzón.

FORMATO DE MENSAJE

El formato del mensaje depende de los objetivos de la facilidad de mensajería y de cuándo tal facilidad ejecuta en un computador único o en un sistema distribuido. En algunos sistemas operativos, los

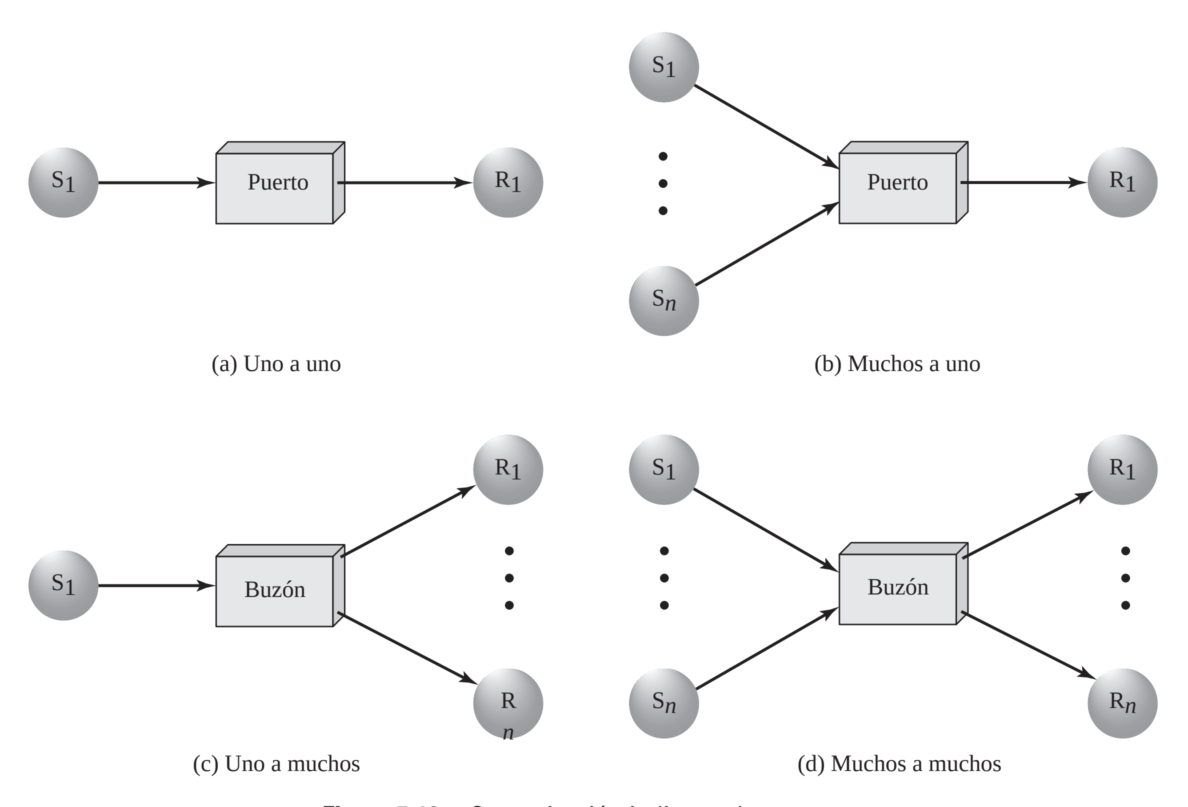


Figura 5.18. Comunicación indirecta de procesos.

diseñadores han preferido mensajes cortos de longitud fija para minimizar la sobrecarga de procesamiento y almacenamiento. Si se va a transferir una gran cantidad de datos, los datos pueden estar dispuestos en un archivo y el mensaje puede simplemente indicar el archivo. Una solución más sencilla es permitir mensajes de longitud variable.

La Figura 5.19 muestra un formato típico de mensaje para un sistema operativo que proporciona mensajes de longitud variable. El mensaje está dividido en dos partes: una cabecera, que contiene información acerca del mensaje, y un cuerpo, que contiene el contenido real del mensaje. La cabecera puede contener una identificación del origen y del destinatario previsto del mensaje, un campo de longitud y un campo de tipo para discriminar entre varios tipos de mensajes. Puede haber también información adicional de control, como un campo puntero, para así poder crear una lista encadenada de mensajes; un número de secuencia, para llevar la cuenta del número y orden de los mensajes intercambiados entre origen y destino; y un campo de prioridad.

DISCIPLINA DE COLA

La disciplina de cola más simple es FIFO, pero puede no ser suficiente si algunos mensajes son más urgentes que otros. Una alternativa es permitir especificar la prioridad del mensaje, en base al tipo de mensaje o por indicación del emisor. Otra alternativa es permitir al receptor inspeccionar la cola de mensajes y seleccionar qué mensaje quiere recibir el siguiente.

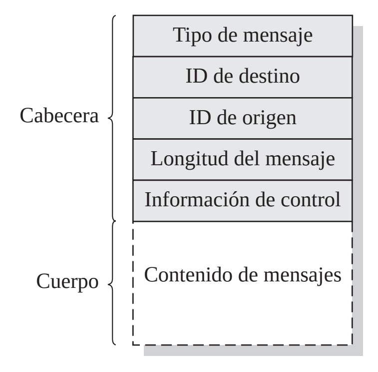


Figura 5.19. Formato general de mensaje.

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
void P(int i)
{
    message carta;
    while (true)
    {
        receive (buzon, carta);
        /* sección crítica */;
        send (buzon, carta);
        /* resto */;
    }
}
void main()
{
        create_mailbox (buzon);
        send (buzon, null);
        paralelos (P(1), P(2), . . ., P(n));
}
```

Figura 5.20. Exclusión mutua usando mensajes.

EXCLUSIÓN MUTUA

La Figura 5.20 muestra un modo de usar el paso de mensajes para conseguir exclusión mutua (compárense las Figuras 5.1, 5.2 y 5.6). Se asume el uso de la primitiva `receive` bloqueante y de la primitiva `send` no bloqueante. Un conjunto de procesos concurrentes comparten un buzón que pueden usar todos los procesos para enviar y recibir. El buzón se inicializa conteniendo un único mensaje de contenido nulo. El proceso que desea entrar en su sección crítica primero intentar recibir un mensaje. Si el buzón está vacío, el proceso se bloquea. Cuando el proceso ha conseguido el mensaje, realiza su

sección crítica y luego devuelve el mensaje al buzón. Así, el mensaje se comporta como un testigo que va pasando de un proceso a otro.

La solución precedente asume que si más de un proceso realiza la operación de recepción concurrentemente, entonces:

- Si hay un mensaje, se le entregará sólo a uno de los procesos y los otros se bloquearán, o
- Si la cola de mensajes está vacía, todos los procesos se bloquearán; cuando haya un mensaje disponible sólo uno de los procesos se activará y tomará el mensaje.

Estos supuestos son prácticamente ciertos en todas las facilidades de paso de mensajes.

Como ejemplo de uso del paso de mensajes, la Figura 5.21 presenta una solución al problema productor/consumidor con *buffer* acotado. Utilizando la exclusión mutua básica que proporciona el

```
/* programa productor consumidor */
const int
    capacidad = /* capacidad de almacenamiento */;
    null = /* mensaje vacío */;
int i;
void productor()
{   message pmsg;
    while (true)
    {
        receive (puedeproducir, pmsg);
        producir();
        receive (puedeconsumir, pmsg);
    }
}
void consumidor()
{   message cmsg;
    while (true)
    {
        receive (puedeconsumir, cmsg);
        consumir();
        receive (puedeproducir, cmsg);
    }
}
void main()
{
    create_mailbox(puedeproducir);
    create_mailbox(puedeconsumir);
    for (int i = 1; i <= capacidad; i++)
        send(puedeproducir, null);
    paralelos (productor, consumidor);
}
```

Figura 5.21. Una solución al problema productor/consumidor con *buffer* acotado usando mensajes.

paso de mensajes, el problema se podría haber resuelto con un algoritmo similar al de la Figura 5.13. En cambio, el programa de la Figura 5.21 aprovecha la ventaja del paso de mensajes para poder transferir datos además de señales. Se utilizan dos buzones. A medida que el productor genera datos, se envían como mensajes al buzón *puedeconsumir*. Tan sólo con que haya un mensaje en el buzón, el consumidor puede consumirlo. Por tanto *puedeconsumir* sirve de *buffer*; los datos en el *buffer* se organizan en una cola de mensajes. El «tamaño» del *buffer* viene determinado por la variable global *capacidad*. Inicialmente, el buzón *puedeproducir* se rellena con un número de mensajes nulos igual a la capacidad del *buffer*. El número de mensajes en *puedeproducir* se reduce con cada producción y crece con cada consumo.

Esta solución es bastante flexible. Puede haber múltiples productores y consumidores, mientras tengan acceso a ambos buzones. El sistema puede incluso ser distribuido, con todos los procesos productores y el buzón *puedeproducir* a un lado y todos los procesos productores y el buzón *puedeconsumir* en el otro.

5.6. EL PROBLEMA DE LOS LECTORES/ESCRITORES

Para el diseño de mecanismos de sincronización y concurrencia, es útil ser capaz de relacionar el problema en concreto con problemas conocidos y ser capaz de probar cualquier solución según su capacidad para resolver estos problemas conocidos. En la literatura, algunos problemas han tomado importancia y aparecen frecuentemente, bien porque son ejemplos de problemas de diseño comunes o bien por su valor educativo. Uno de estos problemas es el productor/consumidor, que ya ha sido explorado. En esta sección, veremos otro problema clásico: el problema lectores/escritores.

El problema lectores/escritores se define como sigue: Hay un área de datos compartida entre un número de procesos. El área de datos puede ser un fichero, un bloque de memoria principal o incluso un banco de registros del procesador. Hay un número de procesos que sólo leen del área de datos (lectores) y otro número que sólo escriben en el área de datos (escritores). Las siguientes condiciones deben satisfacerse.

1. Cualquier número de lectores pueden leer del fichero simultáneamente.
2. Sólo un escritor al tiempo puede escribir en el fichero.
3. Si un escritor está escribiendo en el fichero ningún lector puede leerlo.

Antes de continuar, distingamos este problema de otros dos: el problema general de exclusión mutua y el problema productor/consumidor. En el problema lectores/escritores los lectores no escriben en el área de datos ni los escritores leen del área de datos. Un caso más general, que incluye este caso, es permitir a cualquier proceso leer o escribir en el área de datos. En tal caso, podemos declarar cualquier parte del proceso que accede al área de datos como una sección crítica e imponer la solución general de la exclusión mutua. La razón para preocuparnos por el caso más restrictivo es que es posible una solución más eficiente para este caso y que la solución menos eficiente al problema general es inaceptablemente lenta. Por ejemplo, supóngase que el área compartida es un catálogo de biblioteca. Los usuarios ordinarios de la biblioteca leen el catálogo para localizar un libro. Uno o más bibliotecarios deben poder actualizar el catálogo. En la solución general, cada acceso al catálogo sería tratado como una sección crítica y los usuarios se verían forzados a leer el catálogo de uno en uno. Esto claramente impondría retardos intolerables. Al mismo tiempo, es importante impedir a los escritores interferirse entre sí y también es necesario impedir la lectura mientras la escritura está en curso para impedir que se acceda a información inconsistente.

¿Puede considerarse el problema productor/consumidor simplemente un caso especial del problema lectores/escritores con un único escritor (el productor) y un único lector (el consumidor)? La respuesta es no. El productor no es simplemente un escritor. Él debe leer las posiciones sobre la cola para determinar dónde escribir el siguiente dato y debe determinar si el *buffer* está lleno. De igual modo, el consumidor no es sólo un lector, porque debe ajustar los punteros de la cola para mostrar que ha eliminado una unidad del *buffer*.

Examinemos ahora dos soluciones al problema.

LOS LECTORES TIENEN PRIORIDAD

La Figura 5.22 es una solución utilizando semáforos, que muestra una instancia de cada, un lector y un escritor. El proceso escritor es sencillo. El semáforo `sescr` se utiliza para cumplir la exclusión mutua. Mientras un escritor esté accediendo al área de datos compartidos, ningún otro escritor y ningún lector podrán acceder. El proceso lector también utiliza `sescr` para cumplir la exclusión mutua. No obstante, para permitir múltiples lectores, necesitamos que, cuando no hay lectores leyendo, el primer lector que intenta leer debe esperar en `sescr`. Cuando ya haya al menos un lector leyendo, los siguientes lectores no necesitan esperar antes de entrar. La variable global `cuentalect` se utiliza para llevar la cuenta del número de lectores, y el semáforo `x` se usa para asegurar que `cuentalect` se actualiza adecuadamente.

LOS ESCRITORES TIENEN PRIORIDAD

En la sección previa, los lectores tienen prioridad. Cuando un único lector ha comenzado a acceder al área de datos, es posible que los lectores retengan el control del área de datos mientras quede un lector realizando la lectura. Por tanto, los escritores están sujetos a inanición.

La Figura 5.23 muestra una solución que garantiza que no se le permitirá a ningún lector el acceso al área una vez que al menos un escritor haya declarado su intención de escribir. Para los escritores, los siguientes semáforos y variables se añaden a las ya definidas:

- Un semáforo ***slect*** que inhibe a los lectores mientras haya un único escritor deseando acceder al área de datos.
- Una variable en ***cuentaescr*** que controla el cambio de ***slect***.
- Un semáforo ***y*** que controla la actualización de ***cuentaescr***.

Para los lectores, se necesita un semáforo adicional. No se debe permitir que ocurra una gran cola en ***slect***; de otro modo los escritores no serán capaces de saltar la cola. Por tanto, sólo se le permite a un lector encolarse en ***slect***, cualquier lector adicional se encolará en el semáforo ***z***, inmediatamente antes de esperar en ***slect***. La Tabla 5.5 resume las posibilidades.

En la Figura 5.24 se muestra una solución alternativa, que da prioridad a los escritores y está implementada con el paso de mensajes. En este caso, hay un proceso *controlador* que tiene el acceso al área de datos compartidos. Otros procesos que desean acceder al área de datos envían un mensaje de solicitud al controlador, que concede el acceso respondiendo con un mensaje «OK», e indican que el acceso se ha completado con un mensaje «terminado».

El controlador está equipado con tres buzones, uno por cada uno de los tipos de mensaje que puede recibir.

```
/* programa lectores y escritores */
int cuentalect;
semaphore x = 1, sescr = 1;
void lector()
{
      while (true)
      {
            semWait (x);
            cuentalect++;
            if (cuentalect == 1)
                  semWait (sescr);
            semSignal (x);
            LEERDATO();
            semWait (x);
            cuentalect—;
            if (cuentalect == 0)
                  semSignal (sescr);
            semSignal (x);
      }
}
void escritor()
{
      while (true)
      {
            semWait (sescr);
            ESCRIBIRDATO();
            semSignal (sescr);
      }
}
void main()
{
      cuentalect = 0;
      paralelos (lector, escritor);
}
```

Figura 5.22. Una solución al problema lectores/escritores usando semáforos: los lectores tienen prioridad.

El proceso controlador, para dar prioridad a los escritores, sirve antes los mensajes que solicitan escribir que los mensajes que solicitan leer. Además debe cumplirse la exclusión mutua. Para hacer esto se usa la variable *cuenta*, que se inicializa en algún número mayor que el máximo número posible de lectores. En este ejemplo, se usa el valor 100. La acción del controlador puede resumirse como sigue:

- Si *cuenta* > 0, entonces no hay escritor esperando y puede haber o no lectores activos. Servir primero todos los mensajes «terminado» para limpiar los lectores activos.
- Si *cuenta* = 0, entonces la única solicitud pendiente es una solicitud de escritura. Permitir continuar al escritor y esperar el mensaje «terminado».

```
/* programa lectores y escritores */
int cuentalect, cuentaescr;
semaphore x = 1, y = 1, z = 1, sescr = 1, slect = 1;
void lector()
{
    while (true)
    {
        semWait (z);
            semWait (slect);
                semWait (x);
                    cuentalect++;
                    if (cuentalect == 1)
                        semWait (sescr);
                semSignal (x);
            semSignal (slect);
        semSignal (z);
        LEERDATO();
        semWait (x);
        cuentalect—;
        if (cuentalect == 0)
            semSignal (sescr);
        semSignal (x);
    }
}
void escritor ()
{
    while (true)
    {
        semWait (y);
            cuentaescr++;
            if (cuentaescr == 1)
                semWait (slect);
        semSignal (y);
        semWait (sescr);
        ESCRIBIRDATO();
        semSignal (sescr);
        semWait (y);
            cuentaescr—;
            if (cuentaescr == 0)
                semSignal (slect);
        semSignal (y);
    }
}
void main()
{
    cuentalect = cuentaescr = 0;
    paralelos (lector, escritor);
}
```

Figura 5.23. Una solución al problema lectores/escritores usando semáforos: los escritores tienen prioridad.

Tabla 5.5. Estado de las colas de proceso para el programa de la Figura 5.23.

Sólo lectores en el sistema	• *sescr* establecido • no hay colas
Sólo escritores en el sistema	• *sescr* y *slect* establecidos • los escritores se encolan en *sescr*
Ambos, lectores y escritores, con lectura primero	• *sescr* establecido por lector • *slect* establecido por escritor • todos los escritores se encolan en *sescr* • un lector se encola en *slect* • los otros lectores se encolan en *z*
Ambos, lectores y escritores, con escritura primero	• *sescr* establecido por escritor • *slect* establecido por escritor • los escritores se encolan en *sescr* • un lector se encola en *slect* • los otros lectores se encolan en *z*

- Si *cuenta* < 0, entonces un escritor ha hecho una solicitud y se le está haciendo esperar mientras se limpian todos los lectores activos. Por tanto, sólo deben recibirse mensajes «terminado».

5.7. RESUMEN

Los temas centrales de los sistemas operativos modernos son multiprogramación, multiprocesamiento y procesamiento distribuido. La concurrencia es fundamental en estos temas y fundamental en la tecnología de diseño de sistemas operativos. Cuando múltiples procesos están ejecutando concurrentemente, bien realmente, en el caso de un sistema multiprocesador, o bien virtualmente, en el caso de un sistema multiprogramado de procesador único, surgen cuestiones sobre la resolución de conflictos y la cooperación.

Los procesos concurrentes pueden interactuar de varias maneras. Los procesos que no se percatan unos de otros pueden, sin embargo, competir por recursos, como tiempo de procesador o accesos a dispositivos de entrada/salida. Los procesos pueden percatarse indirectamente unos de otros cuando compartan acceso a un objeto común, como un bloque de memoria principal o un fichero. Finalmente, los procesos pueden ser directamente conscientes unos de otros y cooperar intercambiando información. Los aspectos clave que surgen en estas interacciones son exclusión mutua e interbloqueo.

La exclusión mutua es una condición en la cual hay un conjunto de procesos concurrentes, entre los cuales sólo uno es capaz de acceder a un recurso dado y realizar una función dada en un momento determinado. Las técnicas de exclusión mutua pueden usarse para resolver conflictos, como competencia por recursos o para sincronizar procesos y que así puedan cooperar. Un ejemplo de esto último es el modelo productor/consumidor, en el cual un proceso pone datos en un *buffer* y uno o más procesos extraen datos de ese *buffer*.

Una forma de conseguir la exclusión mutua involucra el uso de instrucciones máquina de propósito específico. Esta solución reduce la sobrecarga pero todavía es ineficiente porque utiliza espera activa.

```
void lector(int i)
{
    message msj;
    while (true)
    {
        msj = i;
        send (quiereleer, msj);
        receive (buzon[i], msj);
        LEERDATO ();
        msj = i;
        send (terminado, msj);
    }
}
void escritor(int j)
{
    message msj;
    while(true)
    {
        msj = j;
        send (quiereescribir, msj);
        receive (buzon[j], msj);
        ESCRIBIRDATO ();
        msj = j;
        send (terminado, msj);
    }
}

void controlador()
{
    while (true)
    {
        if (cuenta > 0)
        {
            if (!vacio (terminado))
            {
                receive (terminado, msj);
                cuenta++;
            }
            else if (!vacio (quiereescribir))
            {
                receive  (quiereescribir,  msj);
                escritor_id = msj.id;
                cuenta = cuenta – 100;
            }
            else if (!vacio (quiereleer))
            {
                receive (quiereleer, msj);
                cuenta—;
                send (msj.id, «OK»);
            }
        }
        if (cuenta == 0)
        {
            send (escritor_id, «OK»);
            receive (terminado, msj);
            cuenta = 100;
        }
        while (cuenta < 0)
        {
            receive (terminado, msj);
            cuenta++;
        }
    }
}
```

Figura 5.24. Una solución al problema lectores/escritores usando el paso de mensajes.

Otro enfoque para conseguir exclusión mutua es proporcionar esos servicios en el sistema operativo. Dos de las técnicas más comunes son los semáforos y las facilidades de mensajes. Los semáforos se utilizan para la señalización entre procesos y se pueden usar fácilmente para conseguir aplicar la disciplina de exclusión mutua. Los mensajes son útiles para hacer cumplir la exclusión mutua y también proporcionan un mecanismo eficiente de comunicación entre procesos.

5.8. LECTURAS RECOMENDADAS

[BEN82] proporciona una explicación muy clara e incluso entretenida de la concurrencia, la exclusión mutua, los semáforos y otros temas relacionados. Un tratamiento más formal, que incluye los sistemas distribuidos se puede encontrar en [BEN90]. [AXFO88] es otro tratado legible y útil; también contiene varios problemas con soluciones desarrolladas. [RAYN86] es una exhaustiva y lúcida colección de algoritmos para la exclusión mutua, cubriendo software (por ejemplo, Dekker) y soluciones hardware, así como semáforos y mensajes. [HOAR85] es un clásico fácil de leer que presenta un enfoque formal a la definición de procesos secuenciales y concurrencia. [LAMP86] es un extenso tratado formal sobre la exclusión mutua. [RUDO90] es una útil ayuda para entender la concurrencia. [BACO03] es un tratado bien organizado sobre la concurrencia. [BIRR89] proporciona una buena introducción práctica a la programación concurrente. [BUHR95] es una exhaustiva investigación sobre los monitores. [KANG98] es un instructivo análisis de 12 políticas de planificación para el problema lectores/escritores.

AXFO88 Axford, T. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. New York: Wiley, 1988.

BACO03 Bacon, J., y Harris, T. *Operating Systems: Concurrent and Distributed Software Design*. Reading, MA: Addison-Wesley, 1998.

BEN82 Ben-Ari, M. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall, 1982.

BEN90 Ben-Ari, M.*Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

BIRR89 Birrell, A. *An Introduction to Programming with Threads*. SRC Research Report 35, Compaq Systems Research Center, Palo Alto, CA, Enero 1989. Disponible en **http://www.research.compaq.com/SRC**.

BUHR95 Buhr, P., y Fortier, M. «Monitor Classification.» *ACM Computing Surveys*, Marzo 1995.

HOAR85 Hoare, C. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall, 1985.

KANG98 Kang, S., y Lee, J. «Analysis and Solution of Non-Preemptive Policies for Scheduling Readers and Writers.» *Operating Systems Review*, Julio 1998.

LAMP86 Lamport, L. «The Mutual Exclusion Problem.» *Journal of the ACM*, Abril 1986.

RAYN86 Raynal, M. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1986.

RUDO90 Rudolph, B. «Self-Assessment Procedure XXI: Concurrency.» *Communications of the ACM*, Mayo 1990.

5.9. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

bloqueante	interbloqueo	sección crítica
concurrencia	monitor	semáforo
condición de carrera	mutex	semáforo binario
corrutina	no bloqueante	semáforo con contador
espera activa	paso de mensajes	semáforo débil
exclusión mutua	proceso concurrente	semáforo fuerte
inanición	recurso crítico	semáforo general

CUESTIONES DE REPASO

5.1. Enumere cuatro aspectos de diseño para los cuales el concepto de concurrencia es relevante.

5.2. ¿En qué tres contextos aparece la concurrencia?

5.3. ¿Cuál es el requisito básico para la ejecución de procesos concurrentes?

5.4. Enumere tres grados de percepción entre procesos y defina brevemente cada uno.

5.5 ¿Cuál es la diferencia entre procesos en competencia y procesos cooperantes?

5.6. Enumere los tres problemas de control asociados con los procesos en competencia y defina brevemente cada uno.

5.7. Enumere las condiciones necesarias para la exclusión mutua.

5.8. ¿Qué operaciones pueden ser realizadas sobre un semáforo?

5.9. ¿Cuál es la diferencia entre semáforos binarios y semáforos generales?

5.10. ¿Cuál es la diferencia entre semáforos fuertes y semáforos débiles?

5.11. ¿Qué es un monitor?

5.12. ¿Cuál es la diferencia entre bloqueante y no bloqueante con respecto a los mensajes?

5.13. ¿Qué condiciones están asociadas generalmente con el problema lectores/escritores?

PROBLEMAS

5.1. Los procesos y los hilos proporcionan una herramienta potente de estructuración para la implementación de programas que serían mucho más complejos como simples programas secuenciales. La corrutina es una construcción concurrente que es instructivo examinar. El propósito de este problema es introducir las corrutinas y compararlas con los procesos. Considere este sencillo problema de [CONW63].

Lea tarjetas de 80 columnas e imprímalas en líneas de 125 caracteres, con los siguientes cambios. Inserte un blanco extra después de la imagen de cada tarjeta y reemplace cada pareja de asteriscos adyacentes (**) que aparezca por el carácter ↑.

a) Desarrolle una solución a este problema como un programa secuencial ordinario. Verá que este programa es difícil de escribir. Las interacciones entre los diversos elementos del programa son dispares debido a la conversión de longitud de 80 a 125; es más, la longitud de la imagen de la tarjeta, después de la conversión, variará dependiendo del número de dobles asteriscos que aparezcan. Una forma de mejorar la claridad y minimizar los posibles errores es escribir la aplicación como tres procedimientos separados. El primer procedimiento lee la imagen de las tarjetas, añade a cada imagen un blanco y escribe el conjunto de caracteres en un fichero temporal. Después de leer todas las tarjetas el segundo procedimiento lee el fichero temporal, hace la sustitución de caracteres y escribe sobre un segundo fichero temporal. El tercer procedimiento lee el contenido del segundo fichero temporal y lo imprime en líneas de 125 caracteres cada una.

b) La solución secuencial no es atractiva por la sobrecarga de E/S y los ficheros temporales. Conway propuso una nueva forma de estructura de programa, la corrutina, que permite escribir la aplicación como tres programas conectados por *buffers* de un carácter (Figura 5.25). En un **procedimiento** tradicional, hay una relación maestro/esclavo entre el procedimiento llamante y el procedimiento llamado. El procedimiento llamante puede ejecutar una llamada desde cualquier punto en el procedimiento; el procedimiento llamado comienza en su punto de entrada y retorna al procedimiento llamante en el punto de la llamada. La **corrutina** exhibe una relación más simétrica. A medida que se realiza cada llamada, la ejecución parte del último punto activo del procedimiento llamado. Dado que en ningún sentido el procedimiento llamante es superior al llamado, no hay retornos. En cambio, una corrutina puede pasar el control a cualquier otra corrutina con la sentencia `resume` (retomar). La primera vez que se invoca a una corrutina, se «retoma» en su punto de entrada. Posteriormente, la corrutina se reactiva en el punto de su propia última sentencia `resume`**.** Nótese que sólo una corrutina del programa puede estar en ejecución en un momento dado y que los puntos de transición están definidos explícitamente en el código, por tanto este no es un ejemplo de procesamiento concurrente. Explique la operación del programa de la Figura 5.25.

c) El programa no contempla la condición de terminación. Asuma que la rutina de E/S LEERTARJETA devuelve el valor cierto si ha puesto una imagen de 80 caracteres en *entrada*; y que devuelve falso en cualquier otro caso. Modifique el programa para incluir esta contingencia. Nótese que la última línea impresa puede, por tanto, contener menos de 125 caracteres.

d) Reescriba la solución como un conjunto de tres procesos usando semáforos.

5.2. Considere un programa concurrente con dos procesos, p y q, definidos como sigue. A, B, C, D y E son sentencias atómicas (indivisibles) arbitrarias. Asuma que el programa principal (que no se muestra) realiza un **paralelos** de los dos procesos.

```
void p()          void q()
{                 {
   A;                D;
   B;                E;
   C;             }
}
```

Muestre todos los posibles entrelazados de la ejecución de los dos procesos precedentes (muéstrelo dando «trazas» de ejecución en términos de sentencias atómicas).

```
char ca, cp;
char entrada[80];
char salida[125];
void leer()
{
    while (true)
    {
        LEERTARJETA (entrada);
        for (int i=0; i < 80; i++)
        {
            ca = entrada [i];
            RESUME filtrar;
        }
        ca = ' ';
        RESUME filtrar;
    }
}
void imprimir()
{
    while (true)
    {
        for (int j = 0; j < 125; j++)
        {
            salida [j] = cp;
            RESUME filtrar;
        }
        IMPRIMIRLINEA (salida);
    }
}
```

```
void filtrar()
{
    while (true)
    {
        if (ca != '*')
        {
            cp = ca;
            RESUME imprimir;
        }
        else
        {
            RESUME leer;
            if (ca == '*')
            {
                cp = '↑';
                RESUME imprimir;
            }
            else
            {
                cp = '*';
                RESUME imprimir;
                cp = ca;
                RESUME imprimir;
            }
        }
        RESUME leer;
    }
}
```

Figura 5.25. Una aplicación de las corrutinas.

5.3. Considere el siguiente programa:

```
const int n = 50;
int cuenta;
void total()
{
    int i;
    for (i = 1; i <= n; i++)
    {
        cuenta ++;
    }
}
```

```
void main()
{
    cuenta = 0;
    paralelos (total (), total ());
    write (cuenta);
}
```

a) Determine los límites inferior y superior correctos del valor final de la variable compartida *cuenta* producida por este programa concurrente. Asuma que los procesos pueden ejecutar a cualquier velocidad relativa y que el valor sólo puede ser incrementado después de haber sido cargado en un registro con una instrucción máquina separada.

b) Suponga que se permite la ejecución en paralelo de un número arbitrario de estos procesos bajo los supuestos del Apartado **a**. ¿Qué efecto tendrá esta modificación en el rango de valores finales de *cuenta*?

5.4. ¿Es siempre menos eficiente (en términos de uso de tiempo de procesador) la espera activa que la espera bloqueante? Explíquelo.

5.5. Considere el siguiente programa:

```
boolean bloqueado[2];
int turno;
void P (int id)
{
    while (true)
    {
        bloqueado[id] = true;
        while (turno != id)
        {
            while (bloqueado[1-id])
                /* no hacer nada */;
            turno = id;
        }
        /* sección crítica */
        bloqueado[id] = false;
        /* resto */
    }
}
void main()
{
    bloqueado[0] = false;
    bloqueado[1] = false;
    turno = 0;
    paralelos (P(0), P(1));
}
```

Esta solución al problema de la exclusión mutua para dos procesos se propone en [HYMA66]. Encuentre un contraejemplo que demuestre que esta solución es incorrecta. Es interesante notar que este código engañó incluso al *Communications of the ACM*.

5.6. Otra solución software a la exclusión mutua es el **algoritmo de la panadería** de Lamport [LAMP74], llamado así porque está basado en la práctica de las panaderías y otras tiendas donde cada cliente recibe un tique numerado cuando llega, que le permite ser servido por turno. El algoritmo es como sigue:

```
boolean eligiendo[n];
int numero[n];
while (true)
{
    eligiendo[i] = true;
    numero[i] = 1 + maximo(numero[], n);
    eligiendo[i] = false;
    for (int j = 0; j < n; j++)
    {
    while (eligiendo[j])
        continue;
    while ((numero[j] != 0) && (numero[j],j) < (numero[i],i))
        continue;
    }
    /* sección crítica */;
    numero[i] = 0;
    /* resto */;
}
```

Los vectores *eligiendo* y *numero* son inicializados a falso y 0 respectivamente. El elemento *i*-ésimo de cada vector puede ser leído y escrito por el proceso *i* pero sólo puede ser leído por otros procesos. La notación $(a,b) < (c,d)$ se define como

$$(a < c) \text{ OR } (a = c \text{ AND } b < d)$$

a) Describa el algoritmo con palabras.

b) Muestre que este algoritmo evita el interbloqueo.

c) Muestre que cumple la exclusión mutua.

5.7. Cuando se utiliza una instrucción máquina especial para proporcionar exclusión mutua a la manera de la Figura 5.2, no hay control sobre cuánto tiempo deberá esperar un proceso antes de conseguir acceso a su región crítica. Idee un algoritmo que utilice la instrucción *testset* para garantizar que cualquier proceso esperando para entrar en su sección crítica lo hará como mucho en n - 1 turnos, donde n es el número de procesos que puede solicitar acceso a la región crítica y un «turno» es un evento consistente en un proceso que sale de su sección crítica y otro proceso al cual se le concede acceso.

5.8. Considere la siguiente definición de semáforos:

```
void semWait(s)
{
```

```
    if (s.cuenta > 0)
    {
        s.cuenta—;
    }
    else
    {
        poner este proceso en s.cola;
        bloquear este proceso;
    }
}
void semSignal (s)
{
    if (hay al menos un proceso bloqueado en el semáforo s)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
    else
        s.cuenta++;
}
```

Compare este conjunto de definiciones con las de la Figura 5.3. Nótese una diferencia: con la definición precedente, un semáforo nunca puede tomar un valor negativo. ¿Hay alguna diferencia entre estos dos conjuntos de definiciones cuando se utilizan en programas? Esto es, ¿podría sustituirse un conjunto por el otro sin alterar el significado del programa?

5.9. Debe ser posible implementar semáforos generales usando semáforos binarios. Podemos usar las operaciones `semWaitB` y `semSignalB` y dos semáforos binarios `pausa` y `mutex.` Considere lo siguiente:

```
void semWait(semaphore s)
{
    semWaitB(mutex);
    s—;
    if (s < 0)
    {
        semSignalB(mutex);
        semWaitB(pausa);
    }
    else
        SemsignalB(mutex);
}
void semSignal(semaphore s);
{
    semWaitB(mutex);
```

```
        s++;
        if (s <= 0)
            semSignalB(pausa);
        semSignalB(mutex);
    }
```

Inicialmente, *s* se pone al valor deseado del semáforo. Cada operación `semWait` decrementa *s* y cada operación `semSignal` incrementa *s*. El semáforo binario `mutex`, que se inicializa a 1, asegura que hay exclusión mutua en la actualización de *s*. El semáforo binario `pausa`, que se inicializa a 0, se usa para bloquear procesos.

Hay un defecto en el programa precedente. Demuestre el defecto y proponga un cambio que lo subsane. *Pista*: suponga dos procesos que llaman a `semWait(s)` cuando s es inicialmente 0 y justo cuando el primero ha realizado `semSignalB(mutex)` pero aún no ha realizado `semWaitB(pausa)`, la segunda llamada a `semWait(s)` avanza hasta el mismo punto. Todo lo que usted tiene que hacer es mover una única línea del programa.

5.10. En 1978, Dijkstra propuso la conjetura de que no hay solución al problema de la exclusión mutua que evite la inanición, aplicable a un número desconocido pero finito de procesos, usando un número finito de semáforos débiles. En 1979, J.M. Morris refutó esta conjetura al publicar un algoritmo que utiliza tres semáforos débiles. El comportamiento del algoritmo puede ser descrito como sigue: si uno o más procesos están esperando en una operación `semWait(S)` y otro proceso está ejecutando `semSignal(S)`, el valor del semáforo S no se modifica y uno de los procesos en espera se desbloquea independientemente de `semWait(S)`. Aparte de los tres semáforos, el algoritmo utiliza dos variables enteras no negativas como contadores del número de procesos dentro de ciertas secciones del algoritmo. Así, los semáforos A y B se inicializan a 1, mientras que el semáforo M y los contadores NA y NM se inicializan a 0. El semáforo de exclusión mutua B protege los accesos a la variable compartida NA. Un proceso que intente entrar en su sección crítica debe cruzar dos barreras representadas por los semáforos A y M. Los contadores NA y NM, respectivamente, contienen el número de procesos listos para cruzar la barrera A y aquéllos que ya han cruzado la barrera A pero todavía no la barrera M. En la segunda parte del protocolo, los NM procesos bloqueados en M entrarán en sus secciones críticas de uno en uno, usando una técnica en cascada similar a la utilizada en la primera parte. Defina el algoritmo que sea conforme con esta descripción.

5.11. El siguiente problema se planteó una vez en un examen:

> Parque Jurásico consiste en un museo y un parque para hacer rutas safari. Hay *m* pasajeros y *n* coches monoplaza. Los pasajeros deambulan por el museo durante un rato y luego hacen cola para dar un paseo en coche por el safari. Cuando hay un coche disponible se monta en él un pasajero y pasea por el parque una cantidad de tiempo aleatoria. Si los *n* vehículos están todos de paseo por el parque con un pasajero abordo, entonces el pasajero que quiere un coche se espera; si un coche está listo para cargar pero no hay pasajeros esperando, entonces el coche se espera. Use semáforos para sincronizar los *m* procesos pasajero con los *n* procesos coche.

El siguiente esqueleto de código fue encontrado garabateado en un papel en el suelo de la sala de examen. Evalúe su corrección. Ignore la sintaxis y las variables no declaradas. Recuerde que P y V se corresponden con `semWait` y `semSignal`.

```
resource Parque_Jurasico()
    sem coche_libre := 0, coche_tomado := 0, coche_ocupado := 0, pasajero_pendiente := 0
```

```
    process pasajero(i := 1 to num_pasajeros)
        do true -> dormir(int(random(1000*tiempo_visita)))
            P(coche_libre); V(coche_tomado); P(coche_ocupado)
            P(pasajero_pendiente)
        od
    end pasajero
    process coche(j := 1 to num_coches)
    do true -> V(coche_libre); P(coche_tomado); V(coche_ocupado)
        dormir(int(random(1000*tiempo_paseo)))
        V(pasajero_pendiente)
    od
    end coche
end Parque_Jurasico
```

5.12. En el comentario sobre la Figura 5.9 y la Tabla 5.3, se indicó que «no cabe simplemente mover la sentencia condicional dentro de la sección crítica (controlada por s) del consumidor, porque esto podría dar lugar a un interbloqueo». Demuestre esto con una tabla similar a la Tabla 5.3.

5.13. Considere la solución al problema productor/consumidor de *buffer* infinito definido en la Figura 5.10. Suponga que tenemos un caso (usual) en el que el productor y el consumidor están ejecutando aproximadamente a la misma velocidad. El escenario podría ser

a) Productor: añadir; `semSignal;` producir; ...; añadir; `semSignal;` producir; ...

Consumidor: consumir; ...; extraer; `semWait;` consumir; ...; extraer; `semWait;` ...

El productor siempre consigue añadir un nuevo elemento al *buffer* y señalar mientras el consumidor consume el elemento previo. El productor siempre está añadiendo a un *buffer* vacío y el consumidor siempre está extrayendo el único dato del *buffer*. Aunque el consumidor nunca se bloquea en el semáforo se están realizando un gran número de llamadas al mecanismo semáforo, creando una considerable sobrecarga.

Construya un nuevo programa que sea más eficiente bajo esas circunstancias. *Pista*: Permita a *n* tomar el valor -1, que significará no sólo que el *buffer* está vacío sino que el consumidor ha detectado este hecho y se va a bloquear hasta que el productor proporcione datos nuevos. La solución no necesita el uso de la variable local *m* que se encuentra en la Figura 5.10.

5.14. Considere la Figura 5.13. ¿Cambiaría el significado del programa si se intercambiase lo siguiente?

a) semWait(e); `semWait(s)`
b) semSignal(s); `semSignal(n)`
c) semWait(n); `semWait(s)`
d) semSignal(s); `semSignal(e)`

5.15. Nótese que en la exposición del problema productor/consumidor con *buffer* finito (Figura 5.12), nuestra definición permite como mucho *n* - 1 entradas en el *buffer*.

a) ¿Por qué es esto?

b) Modifique el algoritmo para corregir esta deficiencia.

5.16. Este problema demuestra el uso de semáforos para coordinar tres tipos de procesos[4]. Santa Claus duerme en su tienda en el Polo Norte y sólo puede ser despertado porque (1) los nueve renos han vuelto todos de sus vacaciones en el Pacífico Sur, o (2) algunos de los elfos tienen dificultades fabricando los juguetes; para que Santa pueda dormir, los elfos sólo pueden despertarle cuando tres de ellos tengan problemas. Cuando tres elfos están solucionando sus problemas cualquier otro elfo que desee visitar a Santa Claus debe esperar a que esos elfos vuelvan. Si Santa Claus se despierta y encuentra a tres elfos esperando en la puerta de su tienda, junto con el último reno que acaba de volver del trópico, Santa Claus tiene decidido que los elfos pueden esperar hasta después de Navidad, porque es más importante tener listo su trineo. (Se asume que los renos no desean abandonar los trópicos y que por tanto están allí hasta el último momento posible). El último reno en volver debe buscar a Santa Claus mientras los otros esperan en un establo calentito antes de ser enganchados al trineo. Resuelva este problema usando semáforos.

5.17. Muestre que el paso de mensajes y los semáforos tienen una funcionalidad equivalente:

a) Implemente el paso de mensajes usando semáforos. *Pista*: haga uso de un área de almacenamiento compartida para mantener los buzones, consistiendo cada uno en un vector con capacidad para un determinado número de mensajes.

b) Implemente un semáforo usando el paso de mensajes. *Pista*: introduzca un proceso de sincronización separado.

[4] Mi gratitud a John Trono del St. Michael's College en Vermont por proporcionar este problema.

CAPÍTULO 6

Concurrencia. Interbloqueo e inanición

◆ ◆

Este capítulo continúa el estudio de la concurrencia examinando dos problemas que dificultan todas las iniciativas para proporcionar procesamiento concurrente: el interbloqueo y la inanición. El capítulo comienza con un estudio de los principios fundamentales de los interbloqueos y los problemas relacionados con la inanición. A continuación, se examinarán las tres estrategias básicas para tratar con el interbloqueo: la prevención, la detección y la predicción. Acto seguido, se revisará uno de los problemas clásicos utilizados para ilustrar tanto la sincronización como el interbloqueo: el problema de los filósofos comensales.

Como en el Capítulo 5, el estudio de este capítulo se limita a considerar la concurrencia y el interbloqueo sobre un único sistema. Las estrategias para tratar con los problemas del interbloqueo distribuido se abordarán en el Capítulo 14.

◆ ◆

6.1. FUNDAMENTOS DEL INTERBLOQUEO

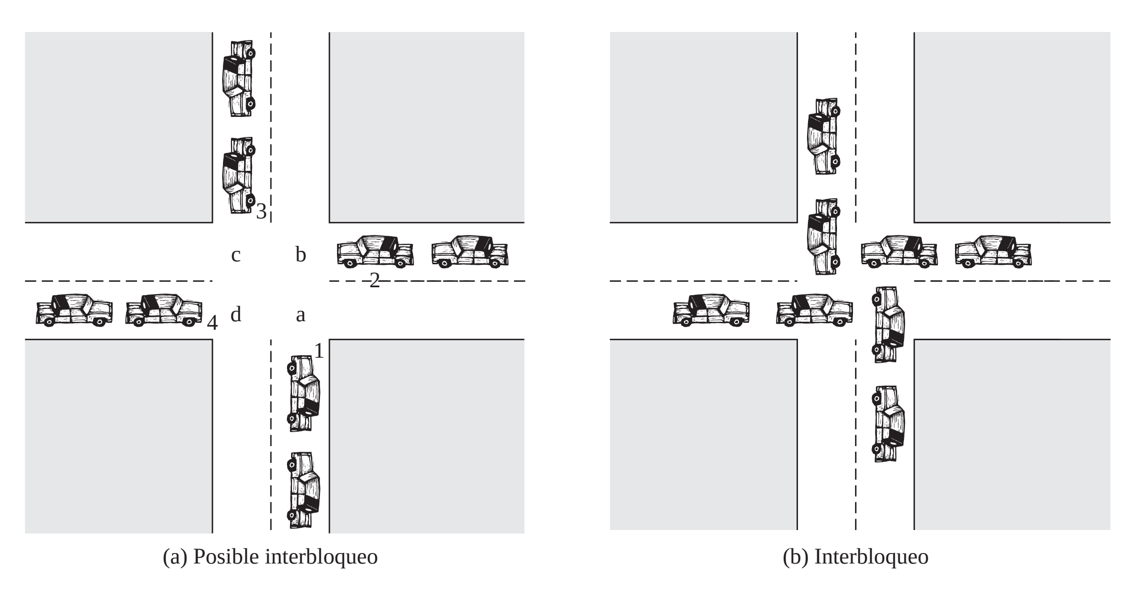
Se puede definir el interbloqueo como el bloqueo *permanente* de un conjunto de procesos que o bien compiten por recursos del sistema o se comunican entre sí. Un conjunto de procesos está interbloqueado cuando cada proceso del conjunto está bloqueado esperando un evento (normalmente la liberación de algún recurso requerido) que sólo puede generar otro proceso bloqueado del conjunto. El interbloqueo es permanente porque no puede producirse ninguno de los eventos. A diferencia de otros problemas que aparecen en la gestión de procesos concurrentes, no hay una solución eficiente para el caso general.

Todos los interbloqueos involucran necesidades conflictivas que afectan a los recursos de dos o más procesos. Un ejemplo habitual es el interbloqueo del tráfico. La Figura 6.1a muestra una situación en la que cuatro coches han llegado aproximadamente al mismo tiempo a una intersección donde confluyen cuatro caminos. Los cuatro cuadrantes de la intersección son los recursos que hay que controlar. En particular, si los cuatro coches desean cruzar la intersección, los requisitos de recursos son los siguientes:

- El coche 1, que viaja hacia el norte, necesita los cuadrantes a y b.
- El coche 2 necesita los cuadrantes b y c.
- El coche 3 necesita los cuadrantes c y d.
- El coche 4 necesita los cuadrantes d y a.

La norma de circulación habitual es que un coche en un cruce de cuatro caminos debería dar preferencia a otro coche que está justo a su derecha. Esta regla funciona si hay sólo dos o tres coches en la intersección. Por ejemplo, si sólo llegan a la intersección los coches que vienen del norte y del oeste, el del norte esperará y el del oeste proseguirá. Sin embargo, si todos los coches llegan aproximadamente al mismo tiempo, cada uno se abstendrá de cruzar la intersección, produciéndose un interbloqueo. Si los cuatro coches olvidan las normas y entran (cuidadosamente) en la intersección, cada uno posee un recurso (un cuadrante) pero no pueden continuar porque el segundo recurso requerido ya se lo ha apoderado otro coche. De nuevo, se ha producido un interbloqueo. Nótese también que debido a que cada coche tiene justo detrás otro coche, no es posible dar marcha atrás para eliminar el interbloqueo.

A continuación, se examina un diagrama de interbloqueo involucrando procesos y recursos del computador. La Figura 6.2 (basada en una incluida en [BACO03]), denominada **diagrama de progreso conjunto**, muestra el progreso de dos procesos compitiendo por dos recursos. Cada proceso ne-



(a) Posible interbloqueo (b) Interbloqueo

Figura 6.1. Ilustración del interbloqueo.

Progreso de Q

Libera A

Libera B

Solicita A

Solicita B

Se requiere A

Se requiere B

1

2

3

4

5

6

P y Q quieren A

Interbloqueo inevitable

P y Q quieren B

Solicita A Solicita B Libera A Libera B

Progreso de P

Se requiere A

Se requiere B

Tanto P como Q quieren el recurso A

Tanto P como Q quieren el recurso B

Región de interbloqueo inevitable

= posible trayectoria de progreso de P y Q
Tramo horizontal de la trayectoria que indica que P está ejecutando y Q esperando.
Tramo vertical de la trayectoria que indica que Q está ejecutando y P esperando

Figura 6.2. Ejemplo de interbloqueo.

cesita el uso exclusivo de ambos recursos durante un cierto periodo de tiempo. Suponga que hay dos procesos, P y Q, que tienen la siguiente estructura general:

Proceso P	**Proceso Q**
...	...
Solicita A	Solicita B
...	...
Solicita B	Solicita A
...	...
Libera A	Libera B
...	...
Libera B	Libera A
...	...

En la Figura 6.2, el eje *x* representa el progreso en la ejecución de P, mientras que el eje *y* representa el de Q. El progreso conjunto de los dos procesos se representa por tanto por una trayectoria que avanza desde el origen en dirección nordeste. En el caso de un sistema uniprocesador, sólo puede ejecutar un proceso cada vez, y la trayectoria consiste en segmentos horizontales y verticales alternados, tal que un segmento horizontal representa un periodo en el que P ejecuta y Q espera, mientras que un segmento vertical representa un periodo en el que Q ejecuta y P espera. La figura muestra áreas en las que tanto P como Q requieren el recurso A (líneas ascendentes); áreas en las que ambos procesos requieren el recurso B (líneas descendentes); y áreas en las que ambos requieren ambos recursos. Debido a que se asume que cada proceso requiere el control exclusivo de un recurso, todas éstas son regiones prohibidas; es decir, es imposible que cualquier trayectoria que represente el progreso de la ejecución conjunta de P y Q entre en una de estas regiones.

La figura muestra seis diferentes trayectorias de ejecución, que se pueden resumir de la siguiente manera:

1. Q adquiere B y, a continuación, A, y, más tarde, libera B y A. Cuando P continúe su ejecución, será capaz de adquirir ambos recursos.
2. Q adquiere B y, a continuación, A. P ejecuta y se bloquea al solicitar A. Q libera B y A. Cuando P continúe su ejecución, será capaz de adquirir ambos recursos.
3. Q adquiere B y, a continuación, P adquiere A. El interbloqueo es inevitable, puesto que cuando la ejecución continúe, Q se bloqueará a la espera de A y P a la de B.
4. P adquiere A y, a continuación, Q adquiere B. El interbloqueo es inevitable, puesto que cuando la ejecución continúe, Q se bloqueará a la espera de A y P a la de B.
5. P adquiere A y, a continuación, B. Q ejecuta y se bloquea al solicitar B. P libera A y B. Cuando Q continúe su ejecución, será capaz de adquirir ambos recursos.
6. P adquiere A y, a continuación, B, y, más tarde, libera A y B. Cuando Q continúe su ejecución, será capaz de adquirir ambos recursos.

El área sombreada en gris en la Figura 6.2, que puede denominarse **región fatal**, está relacionada con el comentario realizado sobre las trayectorias 3 y 4. Si una trayectoria de ejecución entra en esta región fatal, el interbloqueo es inevitable. Nótese que la existencia de una región fatal depende de la lógica de los dos procesos. Sin embargo, el interbloqueo es sólo inevitable si el progreso conjunto de los dos procesos crea una trayectoria que entra en la región fatal.

La aparición de un interbloqueo depende tanto de la dinámica de la ejecución como de los detalles de la aplicación. Por ejemplo, supóngase que P no necesitase ambos recursos al mismo tiempo de manera que los dos procesos tuvieran la siguiente estructura:

Proceso P	**Proceso Q**
...	...
Solicita A	Solicita B
...	...
Libera A	Solicita A
...	...
Solicita B	Libera B
...	...
Libera B	Libera A
...	...

La situación se refleja en la Figura 6.3. Si analiza la figura, el lector se convencerá de que con independencia de la temporización relativa de los dos procesos, no puede ocurrir un interbloqueo.

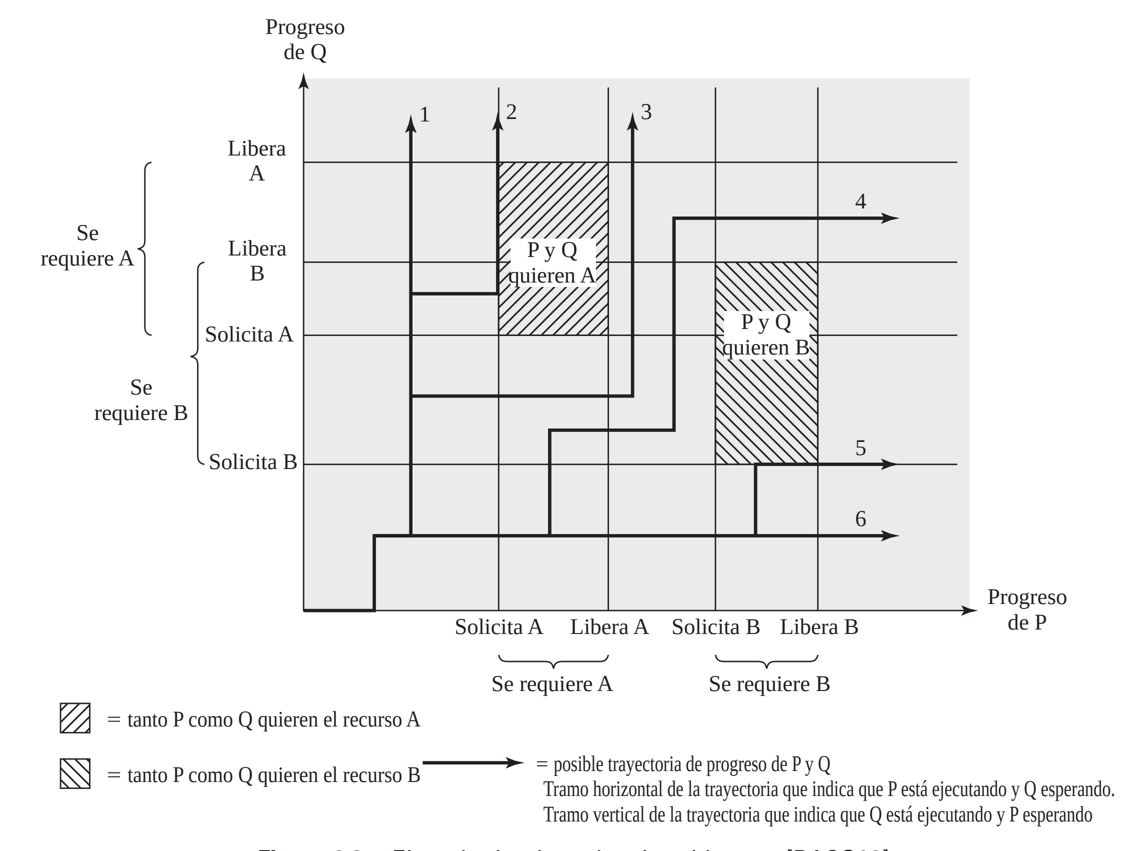


Figura 6.3. Ejemplo donde no hay interbloqueo [BACO03].

Proceso P		Proceso Q	
Paso	**Acción**	**Paso**	**Acción**
p_0	Solicita (D)	q_0	Solicita (C)
p_1	Bloquea (D)	q_1	Bloquea (C)
p_2	Solicita (C)	q_2	Solicita (D)
p_3	Bloquea (C)	q_3	Bloquea (D)
p_4	Realiza función	q_4	Realiza función
p_5	Desbloquea (D)	q_5	Desbloquea (C)
p_6	Desbloquea (C)	q_6	Desbloquea (D)

Figura 6.4. Ejemplo de dos procesos compitiendo por recursos reutilizables.

Como se ha mostrado, se puede utilizar el diagrama de progreso conjunto para registrar la historia de la ejecución de dos procesos que comparten recursos. En los casos donde más de dos procesos pueden competir por el mismo recurso, se requeriría un diagrama con más dimensiones. En cualquier caso, los principios concernientes con las regiones fatales y los interbloqueos permanecerían igual.

RECURSOS REUTILIZABLES

Pueden distinguirse dos categorías de recursos: reutilizables y consumibles. Un recurso reutilizable es aquél que sólo lo puede utilizar de forma segura un proceso en cada momento y que no se destruye después de su uso. Los procesos obtienen unidades del recurso que más tarde liberarán para que puedan volver a usarlas otros procesos. Algunos ejemplos de recursos reutilizables incluyen procesadores, canales de E/S, memoria principal y secundaria, dispositivos, y estructuras de datos como ficheros, bases de datos y semáforos.

Como un ejemplo de recursos reutilizables involucrados en un interbloqueo, considere dos procesos que compiten por el acceso exclusivo a un fichero de disco D y a una unidad de cinta C. En la Figura 6.4 se muestran las operaciones realizadas por los programas implicados. El interbloqueo se produce si cada proceso mantiene un recurso y solicita el otro. Por ejemplo, ocurrirá un interbloqueo si el sistema de multiprogramación intercala la ejecución de los procesos de la siguiente manera:

$$p_0\ p_1\ q_0\ q_1\ p_2\ q_2$$

Puede parecer que se trata de un error de programación más que de un problema del diseñador del sistema operativo. Sin embargo, ya se ha observado previamente que el diseño de programas concurrentes es complejo. Estos interbloqueos se pueden producir, estando su causa frecuentemente empotrada en la compleja lógica del programa, haciendo difícil su detección. Una estrategia para tratar con este interbloqueo es imponer restricciones en el diseño del sistema con respecto al orden en que se pueden solicitar los recursos.

Otro ejemplo de interbloqueo con un recurso reutilizable está relacionado con las peticiones de reserva de memoria principal. Supóngase que el espacio disponible para reservar es de 200 Kbytes y que se produce la secuencia siguiente de peticiones:

P1	**P2**
...	...
Solicita 80 Kbytes;	Solicita 70 Kbytes;
...	...
Solicita 60 Kbytes;	Solicita 80 Kbytes;

El interbloqueo sucede si ambos procesos progresan hasta su segunda petición. Si no se conoce anticipadamente la cantidad de memoria que va a solicitarse, es difícil tratar con este tipo de interbloqueos mediante restricciones en el diseño del sistema. La mejor manera de tratar con este problema es, en realidad, eliminar la posibilidad de que se produzca mediante la utilización de memoria virtual, que se estudiará en el Capítulo 8.

RECURSOS CONSUMIBLES

Un recurso consumible es aquél que puede crearse (producirse) y destruirse (consumirse). Normalmente, no hay límite en el número de recursos consumibles de un determinado tipo. Un proceso productor desbloqueado puede crear un número ilimitado de estos recursos. Cuando un proceso consumidor adquiere un recurso, el recurso deja de existir. Algunos ejemplos de recursos consumibles son las interrupciones, las señales, los mensajes y la información en *buffers* de E/S.

Como un ejemplo de interbloqueo que involucra recursos consumibles, considere el siguiente par de procesos de tal forma que cada proceso intenta recibir un mensaje del otro y, a continuación, le envía un mensaje:

P1	**P2**
...	...
Recibe (P2);	Recibe (P1);
...	...
Envía (P2, M1);	Envía (P1, M2);

Se produce un interbloqueo si la función de recepción (Recibe) es bloqueante (es decir, el proceso receptor se bloquea hasta que se recibe el mensaje). Nuevamente, la causa del interbloqueo es un error de diseño. Estos errores pueden ser bastante sutiles y difíciles de detectar. Además, puede darse una rara combinación de eventos que cause el interbloqueo; así, un programa podría estar usándose durante un periodo considerable de tiempo, incluso años, antes de que realmente ocurra el interbloqueo.

No hay una única estrategia efectiva que pueda tratar todos los tipos de interbloqueo. La Tabla 6.1 resume los elementos fundamentales de las estrategias más importantes que se han desarrollado: prevención, predicción y detección. Se estudiará cada una de ellas, después de que se presenten los grafos de asignación de recursos y, a continuación, las condiciones para el interbloqueo.

Tabla 6.1. Resumen de las estrategias de detección, prevención y predicción de interbloqueos en sistemas operativos [ISLO80].

Estrategia	Política de reserva de recursos	Esquemas alternativos	Principales ventajas	Principales desventajas
Prevención	Conservadora; infrautiliza recursos	Solicitud simultánea de todos los recursos	• Adecuada para procesos que realizan una sola ráfaga de actividad • No es necesaria la expropiación	• Ineficiente • Retrasa la iniciación del proceso • Los procesos deben conocer sus futuros requisitos de recursos
		Expropiación	• Conveniente cuando se aplica a recursos cuyo estado se puede guardar y restaurar fácilmente	• Expropia con más frecuencia de lo necesario
		Ordenamiento de recursos	• Es posible asegurarlo mediante comprobaciones en tiempo de compilación • No necesita cálculos en tiempo de ejecución ya que el problema se resuelve en el diseño del sistema	• Impide solicitudes graduales de recursos
Predicción	A medio camino entre la detección y la prevención	Asegura que existe al menos un camino seguro	• No es necesaria la expropiación	• El SO debe conocer los futuros requisitos de recursos de los procesos • Los procesos se pueden bloquear durante largos periodos
Detección	Muy liberal; los recursos solicitados se conceden en caso de que sea posible	Se invoca periódicamente para comprobar si hay interbloqueo	• Nunca retrasa la iniciación del proceso • Facilita la gestión en línea	• Pérdidas inherentes por expropiación

GRAFOS DE ASIGNACIÓN DE RECURSOS

Una herramienta útil para la caracterización de la asignación de recursos a los procesos es el **grafo de asignación de recursos**, introducido por Holt [HOLT72]. El grafo de asignación de recursos es un grafo dirigido que representa el estado del sistema en lo que se refiere a los recursos y los procesos, de tal forma que cada proceso y cada recurso se representa por un nodo. Una arista del grafo dirigida desde un proceso a un recurso indica que el proceso ha solicitado el recurso pero no se le ha concedido todavía (Figura 6.5a). En el interior de un nodo de recurso, se muestra un punto por cada instancia de ese recurso. Un ejemplo de tipo de recurso que puede tener múltiples instancias es un conjunto de dispositivos de E/S controlados por un módulo de gestión de recursos del sistema operativo. Una arista del grafo dirigida desde un punto de un nodo de un recurso reutilizable hacia un proceso indica que se le ha concedido una petición (Figura 6.5b), es decir, al proceso se le ha asignado una unidad del recurso. Una arista del grafo dirigida desde un punto de un nodo de un recurso consumible hacia un proceso indica que el proceso es el productor de ese recurso.

La Figura 6.5c muestra un ejemplo de interbloqueo. Hay sólo una unidad de cada recurso Ra y Rb. El proceso P1 mantiene Rb y solicita Ra, mientras que P2 mantiene Ra pero pide Rb. La Figura 6.5d tiene la misma topología que la Figura 6.5c, pero no hay interbloqueo porque están disponibles múltiples unidades de cada recurso.

El grafo de asignación de recursos de la Figura 6.6 corresponde a la situación de interbloqueo de la Figura 6.1b. Nótese que en este caso no se trata de una situación sencilla en la cual hay dos proce-

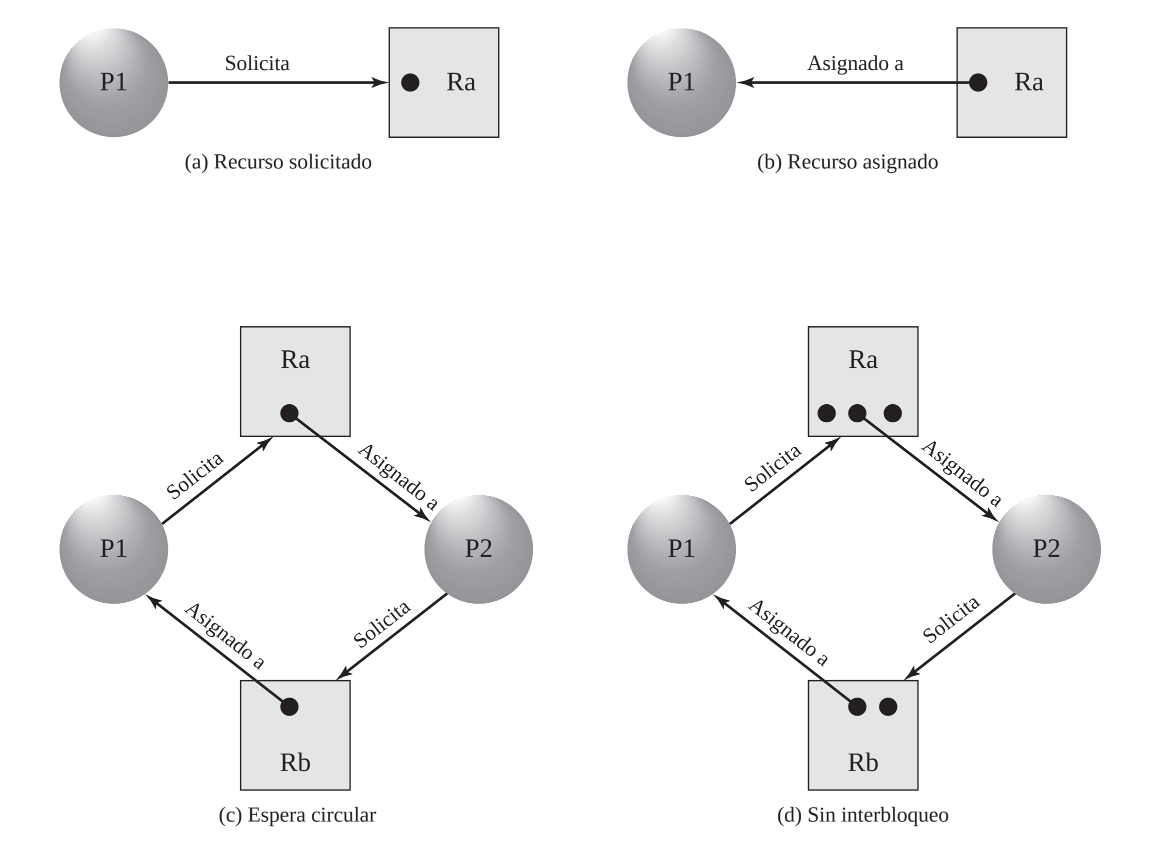


Figura 6.5. Ejemplos de grafos de asignación de recursos.

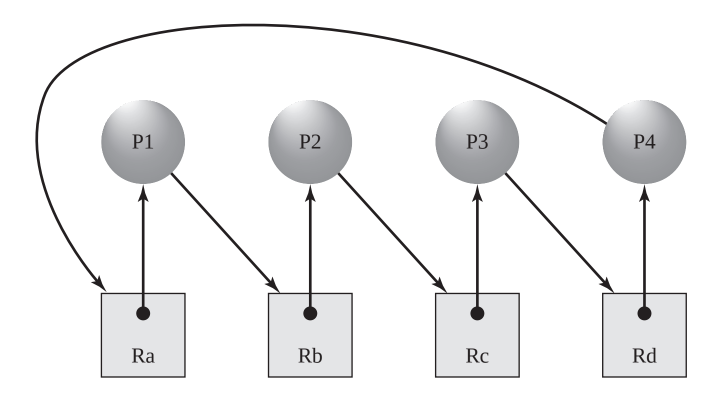


Figura 6.6. Grafo de asignación de recursos correspondiente a la Figura 6.1b.

sos de tal forma que cada uno tiene el recurso que el otro necesita. Más bien, en este caso, hay una cadena circular de procesos y recursos que tiene como resultado un interbloqueo.

LAS CONDICIONES PARA EL INTERBLOQUEO

Deben presentarse tres condiciones de gestión para que sea posible un interbloqueo:

1. **Exclusión mutua.** Sólo un proceso puede utilizar un recurso en cada momento. Ningún proceso puede acceder a una unidad de un recurso que se ha asignado a otro proceso.
2. **Retención y espera.** Un proceso puede mantener los recursos asignados mientras espera la asignación de otros recursos.
3. **Sin expropiación.** No se puede forzar la expropiación de un recurso a un proceso que lo posee.

Por diversos motivos, estas condiciones son realmente deseables. Por ejemplo, se necesita la exclusión mutua para asegurar la coherencia de los resultados y la integridad de una base de datos. Del mismo modo, la expropiación no se debería hacer de una forma arbitraria. Por ejemplo, cuando están involucrados recursos de datos, la expropiación debe implementarse mediante un mecanismo de recuperación mediante retroceso, que restaura un proceso y sus recursos a un estado previo adecuado desde el cual el proceso puede finalmente repetir sus acciones.

Si se cumplen estas tres condiciones se puede producir un interbloqueo, pero aunque se cumplan puede que no lo haya. Para que realmente se produzca el interbloqueo, se requiere una cuarta condición:

4. **Espera circular.** Existe una lista cerrada de procesos, de tal manera que cada proceso posee al menos un recurso necesitado por el siguiente proceso de la lista (como ejemplo, *véase* las Figuras 6.5c y 6.6).

Las tres primeras condiciones son necesarias pero no suficientes para que exista un interbloqueo. La cuarta condición es, realmente, una consecuencia potencial de las tres primeras. Es decir, si se cumplen las tres primeras condiciones, se puede producir una secuencia de eventos que conduzca a una espera circular irresoluble. La espera circular irresoluble es de hecho la definición del interbloqueo. La espera circular enumerada como cuarta condición es irresoluble debido a que se cumplen las

tres primeras condiciones. Por tanto, las cuatro condiciones de forma conjunta constituyen condiciones necesarias y suficientes para el interbloqueo[1].

Para clarificar esta discusión, es útil volver al concepto de diagrama de progreso conjunto, como el que se mostró en la Figura 6.2. Recuerde que se definió una región fatal como una en la que, una vez que los procesos han entrado en la región, estos se verán involucrados en un interbloqueo. Una región fatal sólo existe si se cumplen conjuntamente las tres primeras condiciones anteriormente expuestas. Si no se satisface una o más de estas condiciones, no hay una región fatal y no puede ocurrir un interbloqueo. Por tanto, se trata de condiciones necesarias para el interbloqueo. Para que se produzca el interbloqueo, no debe haber solamente una región fatal, sino también una secuencia de peticiones de recursos que conduzca a la región fatal. Si se cumple la condición de espera circular, se ha entrado, de hecho, en la región fatal. Por tanto, las cuatro condiciones expuestas anteriormente son suficientes para el interbloqueo. Resumiendo:

Posibilidad de interbloqueo	**Existencia de interbloqueo**
1. Exclusión mutua	1. Exclusión mutua
2. Sin expropiación	2. Sin expropiación
3. Retención y espera	3. Retención y espera
	4. Espera circular

Existen tres estrategias para el tratamiento del interbloqueo. En primer lugar, se puede **prevenir** el interbloqueo adoptando una política que elimine una de las condiciones (las 4 condiciones enumeradas previamente). En segundo lugar, se puede **predecir** el interbloqueo tomando las apropiadas decisiones dinámicas basadas en el estado actual de asignación de recursos. En tercer lugar, se puede intentar **detectar** la presencia del interbloqueo (se cumplen las 4 condiciones) y realizar las acciones pertinentes para recuperarse del mismo. A continuación, se estudiarán sucesivamente estas estrategias.

6.2. PREVENCIÓN DEL INTERBLOQUEO

La estrategia de prevención del interbloqueo consiste, de forma simplificada, en diseñar un sistema de manera que se excluya la posibilidad del interbloqueo. Se pueden clasificar los métodos de prevención del interbloqueo en dos categorías. Un método indirecto de prevención del interbloqueo es impedir la aparición de una de las tres condiciones necesarias listadas previamente (las tres primeras). Un método directo de prevención del interbloqueo impide que se produzca una espera circular (cuarta condición). A continuación, se examinan las técnicas relacionadas con las cuatro condiciones.

---

[1] Prácticamente todos los libros de texto simplemente enumeran estas cuatro condiciones como las condiciones necesarias para el interbloqueo, pero esa presentación obscurece algunos de los aspectos más sutiles. La cuarta condición, la espera circular, es fundamentalmente diferente de las otras tres condiciones. Las tres primeras condiciones son decisiones de diseño, mientras que la cuarta es una circunstancia que podría ocurrir dependiendo de la secuencia de peticiones y liberaciones realizada por los procesos involucrados. La asociación entre la espera circular y las tres condiciones necesarias conduce a una inadecuada distinción entre prevención y predicción. *Véase* [SHUB90] y [SHUB03] para una discusión sobre el tema.

EXCLUSIÓN MUTUA

En general, la primera de las cuatro condiciones no puede eliminarse. Si el acceso a un recurso requiere exclusión mutua, el sistema operativo debe proporcionarlo. Algunos recursos, como los ficheros, pueden permitir múltiples accesos de lectura pero acceso exclusivo sólo para las escrituras. Incluso en este caso, puede ocurrir un interbloqueo si más de un proceso requiere permiso de escritura.

RETENCIÓN Y ESPERA

La condición de retención y espera puede eliminarse estableciendo que un proceso debe solicitar al mismo tiempo todos sus recursos requeridos, bloqueándolo hasta que se le puedan conceder simultáneamente todas las peticiones. Esta estrategia es insuficiente en dos maneras. En primer lugar, un proceso puede quedarse esperando mucho tiempo hasta que todas sus solicitudes de recursos puedan satisfacerse, cuando, de hecho, podría haber continuado con solamente algunos de los recursos. En segundo lugar, los recursos asignados a un proceso pueden permanecer inutilizados durante un periodo de tiempo considerable, durante el cual se impide su uso a otros procesos. Otro problema es que un proceso puede no conocer por anticipado todos los recursos que requerirá.

Hay también un problema práctico creado por el uso de una programación modular o una estructura multihilo en una aplicación. La aplicación necesitaría ser consciente de todos los recursos que se solicitarán en todos los niveles o en todos los módulos para hacer una solicitud simultánea.

SIN EXPROPIACIÓN

Esta condición se puede impedir de varias maneras. En primer lugar, si a un proceso que mantiene varios recursos se le deniega una petición posterior, ese proceso deberá liberar sus recursos originales y, si es necesario, los solicitará de nuevo junto con el recurso adicional. Alternativamente, si un proceso solicita un recurso que otro proceso mantiene actualmente, el sistema operativo puede expropiar al segundo proceso y obligarle a liberar sus recursos. Este último esquema impediría el interbloqueo sólo si no hay dos procesos que posean la misma prioridad.

Esta estrategia es sólo práctica cuando se aplica a recursos cuyo estado se puede salvar y restaurar más tarde, como es el caso de un procesador.

ESPERA CIRCULAR

La condición de espera circular se puede impedir definiendo un orden lineal entre los distintos tipos de recursos. Si a un proceso le han asignado recursos de tipo R, posteriormente puede pedir sólo aquellos recursos cuyo tipo tenga un orden posterior al de R.

Para comprobar que esta estrategia funciona correctamente, se puede asociar un índice a cada tipo de recurso, de manera que el recurso R_i precede al R_j en la ordenación si $i < j$. A continuación, supóngase que dos procesos, A y B, están involucrados en un interbloqueo debido a que A ha adquirido R_i y solicitado R_j, y B ha adquirido R_j y solicitado R_i. Esta condición es imposible puesto que implica que $i < j$ y $j < i$.

Como ocurría en el caso de la retención y espera, la prevención de la espera circular puede ser ineficiente, ralentizando los procesos y denegando innecesariamente el acceso a un recurso.

6.3. PREDICCIÓN DEL INTERBLOQUEO

Una estrategia para resolver el problema del interbloqueo que difiere sutilmente de la prevención del interbloqueo es la predicción del interbloqueo[2]. En la **prevención del interbloqueo**, se restringen las solicitudes de recurso para impedir al menos una de las cuatro condiciones de interbloqueo. Esto se realiza o bien indirectamente, impidiendo una de las tres condiciones de gestión necesarias (exclusión mutua, retención y espera, y sin expropiación), o directamente, evitando la espera circular. Esto conlleva un uso ineficiente de los recursos y una ejecución ineficiente de los procesos. La **predicción del interbloqueo**, por otro lado, permite las tres condiciones necesarias pero toma decisiones razonables para asegurarse de que nunca se alcanza el punto del interbloqueo. De esta manera, la predicción permite más concurrencia que la prevención. Con la predicción del interbloqueo, se decide dinámicamente si la petición actual de reserva de un recurso, si se concede, podrá potencialmente causar un interbloqueo. La predicción del interbloqueo, por tanto, requiere el conocimiento de las futuras solicitudes de recursos del proceso.

En esta sección se describen dos técnicas para predecir el interbloqueo:

- No iniciar un proceso si sus demandas podrían llevar al interbloqueo.
- No conceder una petición adicional de un recurso por parte de un proceso si esta asignación podría provocar un interbloqueo.

DENEGACIÓN DE LA INICIACIÓN DEL PROCESO

Considere un sistema con n procesos y m tipos diferentes de recursos. En el mismo se definen los siguientes vectores y matrices:

Recursos = $\mathbf{R} = (R_1, R_2, ..., R_m)$	cantidad total de cada recurso en el sistema
Disponibles = $\mathbf{D} = (D_1, D_2, ..., D_m)$	cantidad total de cada recurso no asignada a ningún proceso
Necesidad = $\mathbf{N} = \begin{vmatrix} N_{11} & N_{12} & ... & N_{1m} \\ N_{21} & N_{22} & ... & N_{2m} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ N_{n1} & N_{n2} & ... & N_{nm} \end{vmatrix}$	N_{ij} = necesidades del proceso i con respecto al recurso j
Asignación = $\mathbf{A} = \begin{vmatrix} A_{11} & A_{12} & ... & A_{1m} \\ A_{21} & A_{22} & ... & A_{2m} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ A_{n1} & A_{n2} & ... & A_{nm} \end{vmatrix}$	A_{ij} = asignación actual al proceso i con respecto al recurso j

[2] El término *predicción* es un poco confuso. De hecho, se podrían considerar las estrategias estudiadas en esta sección como ejemplos de prevención del interbloqueo debido a que impiden efectivamente la aparición del interbloqueo.

La matriz de Necesidad proporciona los requisitos máximos de cada proceso con respecto a cada recurso, estando una fila dedicada a cada proceso. Esta información debe declararse con antelación por el proceso para que la predicción del interbloqueo funcione. De modo similar, la matriz de Asignación proporciona la asignación actual a cada proceso. Se cumplen las siguientes relaciones:

1. $R_j = D_j + \sum_{i=1}^{n} A_{ij}$, para todo j — Todos los recursos están disponibles o asignados.
2. $N_{ij} \leq R_j$, para todo i, j — Ningún proceso puede necesitar más de la cantidad total de recursos existentes en el sistema.
3. $A_{ij} \leq N_{ij}$, para todo i, j — Ningún proceso tiene asignados más recursos de cualquier tipo que sus necesidades originales de ese recurso.

Con estas cantidades definidas, se puede establecer una política de predicción del interbloqueo que rechace iniciar un nuevo proceso si sus requisitos de recursos pudiesen conducir al interbloqueo. Se inicia un nuevo proceso P_{n+1} sólo si:

$$R_j \geq N_{(n+1)j} + \sum_{i=1}^{n} N_{ij} \qquad \text{para todo } j$$

Es decir, sólo puede iniciarse un proceso si se pueden satisfacer las necesidades máximas de todos los procesos actuales más las del nuevo proceso. Esta estrategia está lejos de ser óptima, debido a que asume el peor caso: todos los procesos solicitarán sus necesidades máximas simultáneamente.

DENEGACIÓN DE ASIGNACIÓN DE RECURSOS

La estrategia de la denegación de asignación de recursos, denominada algoritmo del banquero[3], se propuso por primera vez en [DIJK65]. Se comenzará definiendo los conceptos de estado y de estado seguro. Considere un sistema con un número fijo de procesos y de recursos. En un determinado momento un proceso puede tener cero o más recursos asignados. El **estado** del sistema refleja la asignación actual de recursos a procesos. Por tanto, el estado consiste en los dos vectores, Recursos y Disponibles, y las dos matrices, Necesidad y Asignación, definidas anteriormente. Un **estado seguro** es aquél en el que hay al menos una secuencia de asignación de recursos a los procesos que no implica un interbloqueo (es decir, todos los procesos pueden ejecutarse al completo). Un **estado inseguro** es, evidentemente, un estado que no es seguro.

El siguiente ejemplo muestra estos conceptos. La Figura 6.7a muestra el estado de un sistema que consta de cuatro procesos y tres recursos. La cantidad total de recursos R1, R2, y R3 son de 9, 3 y 6 unidades, respectivamente. En el estado actual se han realizado asignaciones a los cuatro procesos, dejando disponibles 1 unidad de R2 y 1 unidad de R3. La pregunta es: ¿es un estado seguro? Para responder a esta cuestión, se plantea una pregunta intermedia: ¿alguno de los cuatro procesos

---

[3] Dijkstra usó este nombre debido a la analogía de este problema con uno que se da en la banca, donde los clientes que desean un préstamo de dinero se corresponden con los procesos y el dinero para prestar se corresponde con los recursos. Enunciado como un problema bancario, el problema sería como se especifica a continuación. El banco tiene una reserva limitada de dinero para prestar y una lista de clientes, cada uno con una línea de crédito. En un momento dado, un cliente puede optar por solicitar un préstamo de una cantidad que corresponda con una parte de su línea de crédito, y no hay garantía de que haga un reembolso hasta después de haber obtenido la cantidad máxima del préstamo. El banquero puede rechazar un préstamo a un cliente si hay riesgo de que el banco se quede sin fondos suficientes para satisfacer solicitudes de préstamo posteriores que permitirán que los clientes realicen finalmente un reembolso.

puede ejecutarse por completo con los recursos disponibles? Es decir, ¿puede satisfacerse con los recursos disponibles la diferencia entre los requisitos máximos y la asignación actual de algún proceso?

En términos de las matrices y vectores presentados previamente, la condición que debe cumplirse para el proceso *i* es la siguiente:

$$N_{ij} - A_{ij} \leq D_j, \text{ para todo } j$$

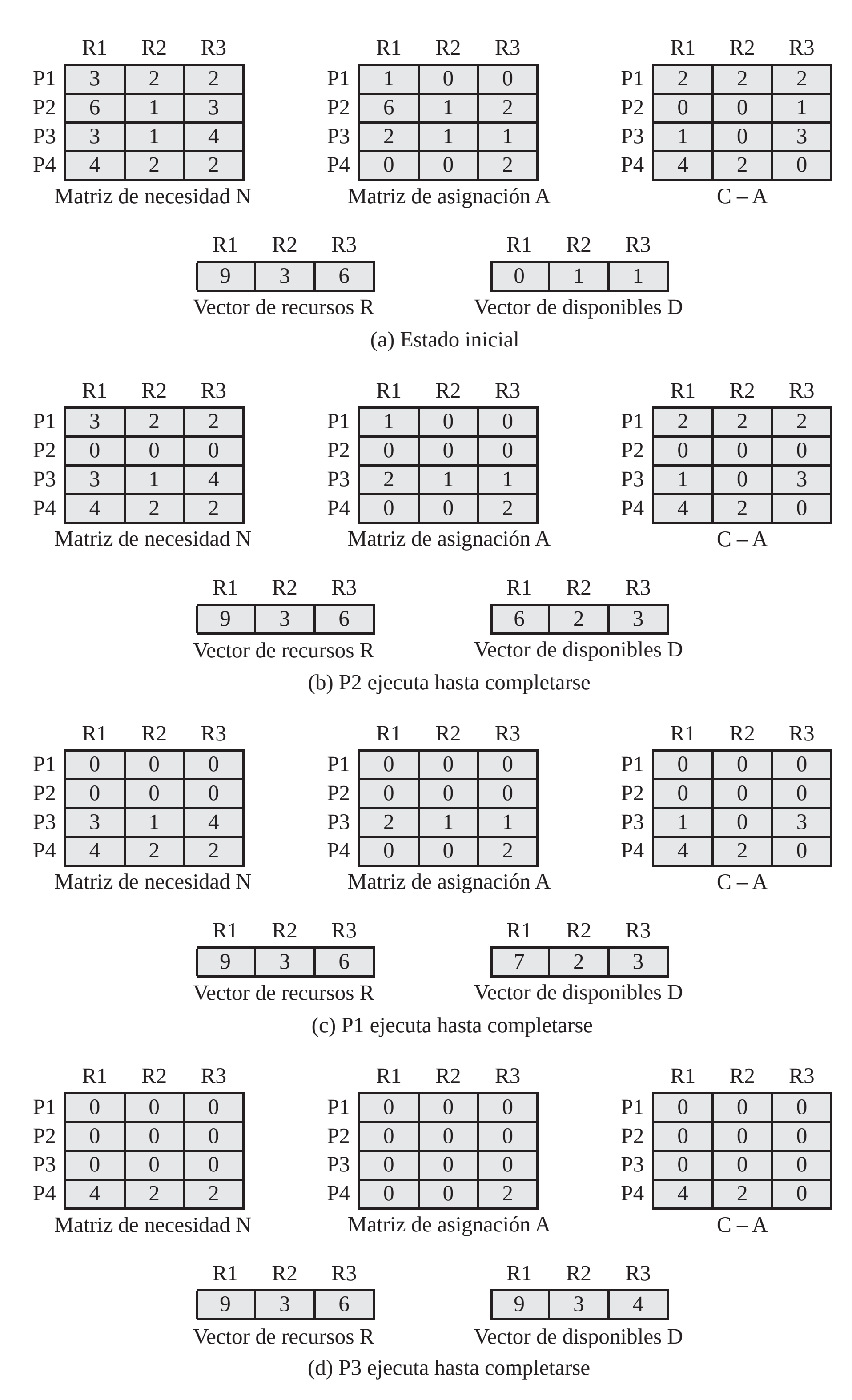
Claramente, esto no es posible en el caso de P1, que tiene sólo una unidad de R1 y requiere 2 unidades adicionales, otras 2 de R2, así como 2 unidades de R3. Sin embargo, asignando una unidad de R3 al proceso P2, éste logra tener asignados sus recursos máximos requeridos y puede ejecutarse por completo. Asuma que esto se lleva a cabo. Cuando se complete P2, podrá retornar sus recursos al conjunto de recursos disponibles. El estado resultante se muestra en la Figura 6.7b. En ese momento, se repetiría la pregunta de si cualquiera de los procesos restantes puede completarse. En este caso, todos los procesos restantes podrían completarse. Supóngase, que se elige P1, asignándole los recursos requeridos, se completa P1, y devuelve todos sus recursos al conjunto de disponibles. El estado resultante se muestra en la Figura 6.7c. A continuación, se puede completar P3, alcanzándose el estado de la Figura 6.7d. Finalmente, se puede completar P4. En ese instante, todos los procesos se han ejecutado por completo. Por tanto, el estado definido en la Figura 6.7a es seguro.

Estos conceptos sugieren la siguiente estrategia de predicción del interbloqueo, que asegura que el sistema de procesos y recursos está siempre en un estado seguro. Cuando un proceso solicite un conjunto de recursos, supóngase que se concede la petición, actualice el estado del sistema en consecuencia, y determine si el resultado es un estado seguro. En caso afirmativo, se concede la petición. En caso contrario, se bloquea el proceso hasta que sea seguro conceder la petición.

Considérese el estado definido en la Figura 6.8a. Supóngase que P2 solicita una unidad adicional de R1 y otra de R3. Si se asume que se concede la petición, el estado resultante es el de la Figura 6.7a, que ya se ha comprobado previamente que es un estado seguro. Por tanto, es seguro conceder la petición. Retornando al estado de la Figura 6.8a y suponiendo que P1 solicita una unidad de R1 y otra de R3; si se asume que se concede la petición, el estado resultante es el representado en la Figura 6.8b. ¿Es un estado seguro? La respuesta es que no, debido a que cada proceso necesitará al menos una unidad adicional de R1, y no hay ninguna disponible. Por tanto, basándose en la predicción del interbloqueo, la solicitud de P1 se denegaría y P1 se debería bloquear.

Es importante resaltar que la Figura 6.8b no es un estado de interbloqueo. Sólo indica la posibilidad del interbloqueo. Es posible, por ejemplo, que si P1 ejecutara a partir de este estado, liberaría posteriormente una unidad de R1 y una de R3 antes de volver a necesitar estos recursos de nuevo. Si esto sucediera, el sistema retornaría a un estado seguro. Por tanto, la estrategia de predicción de interbloqueo no predice el interbloqueo con certeza; sólo anticipa la posibilidad del interbloqueo y asegura que no haya tal posibilidad.

La Figura 6.9 da una versión abstracta de la lógica de predicción del interbloqueo. El algoritmo principal se muestra en la parte (b). La estructura de datos `estado` define el estado del sistema, mientras que `peticion[*]` es un vector que define los recursos solicitados por el proceso *i*. En primer lugar, se hace una prueba para asegurarse de que la petición no excede las necesidades originales del proceso. Si la petición es válida, el siguiente paso es determinar si es posible satisfacerla (es decir, hay suficientes recursos disponibles). Si no es posible, se suspende la ejecución del proceso. En caso de que lo sea, el paso final es determinar si es seguro satisfacer la petición. Para hacer esto, se asignan provisionalmente los recursos al proceso *i* para formar el `nuevo_estado`. A continuación, se realiza una comprobación de si el estado es seguro utilizando el algoritmo de la Figura 6.9c.



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Vector de recursos R

R1	R2	R3
0	1	1

Vector de disponibles D

(a) Estado inicial

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Vector de recursos R

R1	R2	R3
6	2	3

Vector de disponibles D

(b) P2 ejecuta hasta completarse

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Vector de recursos R

R1	R2	R3
7	2	3

Vector de disponibles D

(c) P1 ejecuta hasta completarse

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

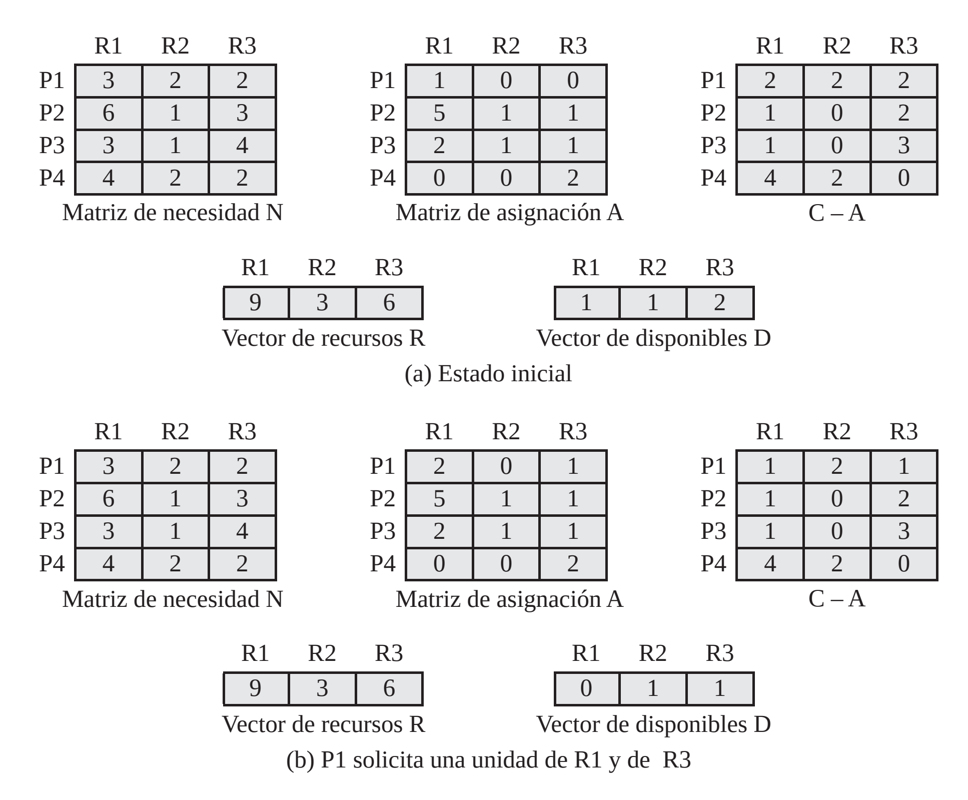
Vector de recursos R

R1	R2	R3
9	3	4

Vector de disponibles D

(d) P3 ejecuta hasta completarse

Figura 6.7. Determinación de un estado seguro.



	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Vector de recursos R

R1	R2	R3
1	1	2

Vector de disponibles D

(a) Estado inicial

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Vector de recursos R

R1	R2	R3
0	1	1

Vector de disponibles D

(b) P1 solicita una unidad de R1 y de R3

Figura 6.8. Determinación de un estado inseguro.

La predicción del interbloqueo tiene la ventaja de que no es necesario expropiar a los procesos ni retroceder su ejecución, como ocurre con la detección del interbloqueo, y es menos restrictivo que la prevención del interbloqueo. Sin embargo, tiene varias restricciones de uso:

- Deben establecerse por anticipado los requisitos máximos de recursos de cada proceso.
- Los procesos involucrados deben ser independientes, es decir, el orden en el que se ejecutan no debe estar restringido por ningún requisito de sincronización.
- Debe haber un número fijo de recursos que asignar.
- Ningún proceso puede terminar mientras mantenga recursos.

6.4. DETECCIÓN DEL INTERBLOQUEO

Las estrategias de prevención de interbloqueo son muy conservadoras: resuelven el problema del interbloqueo limitando el acceso a los recursos e imponiendo restricciones a los procesos. En el extremo contrario, la estrategia de detección del interbloqueo no limita el acceso a los recursos ni restringe las acciones de los procesos. Con la detección del interbloqueo, los recursos pedidos se conceden a los procesos siempre que sea posible. Periódicamente, el sistema operativo realiza un algoritmo que le permite detectar la condición de espera circular descrita anteriormente en la condición (4) e ilustrada en la Figura 6.6.

```
struct estado
{
      int recursos[m];
      int disponibles[m];
      int necesidad[n][m];
      int asignacion[n][m];
}
```

(a) estructuras de datos globales

```
if (asignacion [i,*] + peticion [*] > necesidad [i,*])
      < error >;                                         /* petición total > necesidad */
else if (peticion [*] > disponibles [*])
      < suspender al proceso >;
else                                                     /* simular asignación */
{
      < definir nuevo_estado como:
      asignacion [i,*] = asignacion [i,*] + peticion [*];
      disponibles [*] = disponibles [*] - peticion [*] >;
}
if (seguro(nuevo_estado))
      < llevar a cabo la asignación >;
else
{
      < restaurar el estado original >;
      < suspender al proceso >;
}
```

(b) algoritmo de asignación de recursos

```
boolean seguro (estado E)
{
      int disponibles_actual[m];
      proceso resto[<número de procesos>];
disponibles_actual = disponibles;
resto = {todos los procesos};
      posible = verdadero;
      while (posible)
      {
            <encontrar un proceso Pk en resto tal que
                  necesidad [k,*] - asignacion [k,*] <= disponibles_actual;>
            if (encontrado)                              /* simular ejecución de Pk */
            {
                  disponibles_actual = disponibles_actual + asignacion [k,*];
                  resto = resto – {Pk};
            }
            else
                  posible = falso;
      }
      return (resto == null);
}
```

(c) algoritmo para comprobar si el estado es seguro (algoritmo del banquero)

Figura 6.9. Lógica para la predicción del interbloqueo.

ALGORITMO DE DETECCIÓN DEL INTERBLOQUEO

La comprobación de si hay interbloqueo se puede hacer con tanta frecuencia como una vez por cada petición de recurso o, con menos frecuencia, dependiendo de la probabilidad de que ocurra un interbloqueo. Realizar la comprobación por cada petición de recurso tiene dos ventajas: conlleva una detección temprana y el algoritmo es relativamente sencillo debido a que está basado en cambios graduales del estado del sistema. Por otro lado, estas comprobaciones frecuentes consumen un considerable tiempo del procesador.

Un algoritmo usual para la detección del interbloqueo es el que se describe en [COFF71]. En el mismo se utilizan la matriz de Asignación y el vector de Disponibles descritos en la sección previa. Además, se define una matriz de solicitud **S** tal que S_{ij} representa la cantidad de recursos de tipo *j* solicitados por el proceso *i*. El algoritmo actúa marcando los procesos que no están en un interbloqueo. Inicialmente, todos los procesos están sin marcar. A continuación, se llevan a cabo los siguientes pasos:

1. Se marca cada proceso que tenga una fila de la matriz de Asignación completamente a cero.
2. Se inicia un vector temporal **T** asignándole el vector Disponibles.
3. Se busca un índice *i* tal que el proceso *i* no esté marcado actualmente y la fila *i*-ésima de **S** sea menor o igual que **T**. Es decir, $S_{ik} \leq T_k$, para $1 \leq k \leq m$. Si no se encuentra ninguna fila, el algoritmo termina.
4. Si se encuentra una fila que lo cumpla, se marca el proceso *i* y se suma la fila correspondiente de la matriz de asignación a **T**. Es decir, se ejecuta $T_k = T_k + A_{ik}$, para $1 \leq k \leq m$. A continuación, se vuelve al tercer paso.

Existe un interbloqueo si y sólo si hay procesos sin marcar al final del algoritmo. Cada proceso sin marcar está en un interbloqueo. La estrategia de este algoritmo es encontrar un proceso cuyas peticiones de recursos puedan satisfacerse con los recursos disponibles, y, a continuación, asumir que se conceden estos recursos y el proceso se ejecuta hasta terminar y libera todos sus recursos. El algoritmo, a continuación, busca satisfacer las peticiones de otro proceso. Nótese que este algoritmo no garantiza la prevención del interbloqueo; esto dependerá del orden en que se concedan las peticiones futuras. Su labor es determinar si existe actualmente un interbloqueo.

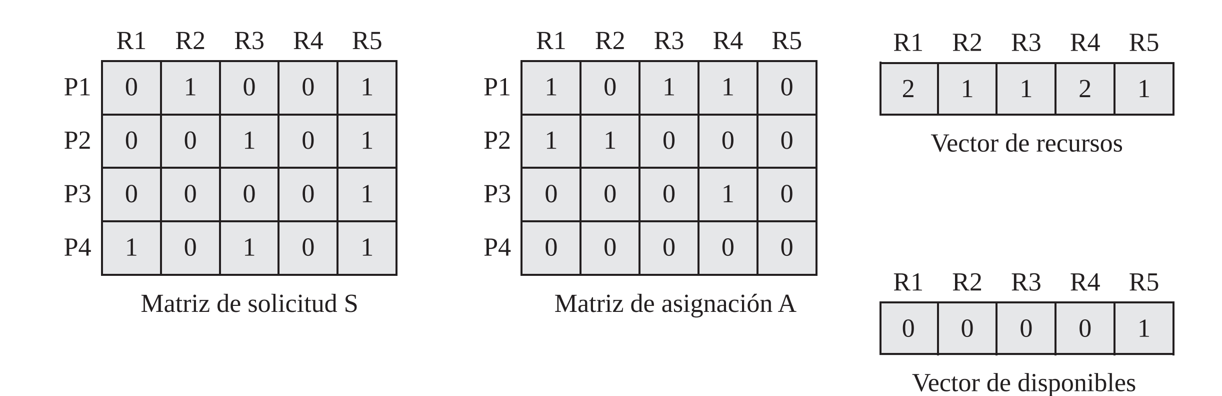
Se puede utilizar la Figura 6.10 para mostrar el algoritmo de detección del interbloqueo. El algoritmo actúa de la siguiente manera:

1. Marca P4, porque no tiene recursos asignados.
2. Fija **T** = (0 0 0 0 1).
3. La petición del proceso P3 es menor o igual que **T**, así que se marca P3 y ejecuta **T** = **T** + (0 0 0 1 0) = (0 0 0 1 1).
4. Ningún otro proceso sin marcar tiene una fila de **S** que sea menor o igual a **T**. Por tanto, el algoritmo termina.

El algoritmo concluye sin marcar P1 ni P2, indicando que estos procesos están en un interbloqueo.

RECUPERACIÓN

Una vez que se ha detectado el interbloqueo, se necesita alguna estrategia para recuperarlo. Las siguientes estrategias, listadas en orden de sofisticación creciente, son posibles:



	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Matriz de solicitud S

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Matriz de asignación A

R1	R2	R3	R4	R5
2	1	1	2	1

Vector de recursos

R1	R2	R3	R4	R5
0	0	0	0	1

Vector de disponibles

Figura 6.10. Ejemplo de detección del interbloqueo.

1. Abortar todos los procesos involucrados en el interbloqueo. Esta es, se crea o no, una de las más usuales, si no la más, solución adoptada en los sistemas operativos.
2. Retroceder cada proceso en interbloqueo a algún punto de control (*checkpoint*) previamente definido, y rearrancar todos los procesos. Esto requiere que se implementen en el sistema mecanismos de retroceso y rearranque. El riesgo de esta técnica es que puede repetirse el interbloqueo original. Sin embargo, el indeterminismo del procesamiento concurrente puede asegurar que probablemente esto no suceda.
3. Abortar sucesivamente los procesos en el interbloqueo hasta que éste deje de existir. El orden en que seleccionan los procesos para abortarlos debería estar basado en algunos criterios que impliquen un coste mínimo. Después de cada aborto, se debe invocar de nuevo el algoritmo de detección para comprobar si todavía existe el interbloqueo.
4. Expropiar sucesivamente los recursos hasta que el interbloqueo deje de existir. Como en el tercer punto, se debería utilizar una selección basada en el coste, y se requiere una nueva invocación del algoritmo de detección después de cada expropiación. Un proceso al que se le ha expropiado un recurso debe retroceder a un punto anterior a la adquisición de ese recurso.

Para los Puntos (3) y (4), el criterio de selección podría ser uno de los siguientes. Se elige el proceso con:

- la menor cantidad de tiempo de procesador consumida hasta ahora
- la menor cantidad de salida producida hasta ahora
- el mayor tiempo restante estimado
- el menor número total de recursos asignados hasta ahora
- la menor prioridad

Algunas de estas cantidades son más fáciles de medir que otras. El tiempo restante estimado es particularmente de difícil aplicación. Además, con excepción del criterio basado en la prioridad, no hay ninguna otra indicación del «coste» para el usuario, en contraposición con el coste para el sistema como un todo.

6.5. UNA ESTRATEGIA INTEGRADA DE TRATAMIENTO DEL INTERBLOQUEO

Como sugiere la Tabla 6.1, hay ventajas y desventajas en todas las estrategias para el tratamiento del interbloqueo. En vez de intentar diseñar una solución en el sistema operativo que utilice una sola de estas estrategias, podría ser más eficiente usar estrategias diferentes en distintas situaciones. [HOWA73] sugiere la siguiente técnica:

- Agrupar los recursos en diversas clases de recursos diferentes.
- Utilizar la estrategia del orden lineal definida previamente para prevenir la espera circular impidiendo los interbloqueos entre clases de recursos.
- Dentro de una clase de recursos, usar el algoritmo que sea más apropiado para esa clase.

Como un ejemplo de esta técnica, considere las siguientes clases de recursos:

- **Espacio de intercambio.** Bloques de memoria en almacenamiento secundario utilizados al expulsar los procesos.
- **Recursos del proceso.** Dispositivos asignables, como dispositivos de cinta y ficheros.
- **Memoria principal.** Asignable a los procesos en páginas o segmentos.
- **Recursos internos.** Como canales de E/S.

El orden de la lista anterior representa el orden en el que se asignan los recursos. Es un orden razonable, considerando la secuencia de pasos que puede seguir un proceso durante su tiempo de vida. Dentro de cada clase, se podrían utilizar las siguientes estrategias:

- **Espacio de intercambio.** La prevención de los interbloqueos, obligando a que se asignen al mismo tiempo todos los recursos necesitados que vayan a usarse, como en la estrategia de prevención de la retención y espera. Esta estrategia es razonable si se conocen los requisitos máximos de almacenamiento, lo cual frecuentemente es cierto. La predicción también es una posibilidad factible.
- **Recursos del proceso.** La predicción será usualmente efectiva para esta categoría, porque es razonable esperar que los procesos declaren anticipadamente los recursos de esta clase que requerirán. La prevención mediante ordenamiento de recursos es también posible para esta clase.
- **Memoria principal.** La prevención por expropiación parece la estrategia más apropiada para la memoria principal. Cuando se expropia a un proceso, simplemente es expulsado a memoria secundaria, liberando el espacio para resolver el interbloqueo.
- **Recursos internos.** Puede utilizarse la prevención mediante el ordenamiento de recursos.

6.6. EL PROBLEMA DE LOS FILÓSOFOS COMENSALES

En esta sección se trata el problema de los filósofos comensales, presentado por Dijkstra [DIJK71]. Cinco filósofos viven en una casa, donde hay una mesa preparada para ellos. Básicamente, la vida de cada filósofo consiste en pensar y comer, y después de años de haber estado pensando, todos los filósofos están de acuerdo en que la única comida que contribuye a su fuerza mental son los espaguetis. Debido a su falta de habilidad manual, cada filósofo necesita dos tenedores para comer los espaguetis.

La disposición para la comida es simple (Figura 6.11): una mesa redonda en la que está colocado un gran cuenco para servir espaguetis, cinco platos, uno para cada filósofo, y cinco tenedores. Un filósofo que quiere comer se dirige a su sitio asignado en la mesa y, utilizando los dos tenedores situados a cada lado del plato, toma y come algunos espaguetis. El problema: diseñar un ritual (algoritmo) que permita a los filósofos comer. El algoritmo debe satisfacer la exclusión mutua (no puede haber dos filósofos que puedan utilizar el mismo tenedor a la vez) evitando el interbloqueo y la inanición (en este caso, el término tiene un sentido literal, además de algorítmico).

Este problema puede que no parezca importante o relevante en sí mismo. Sin embargo, muestra los problemas básicos del interbloqueo y la inanición. Además, intenta desarrollar soluciones que revelan muchas de las dificultades de la programación concurrente (como ejemplo, *véase* [GING90]). Asimismo, el problema de los filósofos comensales puede considerarse como representativo de los problemas que tratan la coordinación de recursos compartidos, que puede ocurrir cuando una aplicación incluye hilos concurrentes en su ejecución. Por consiguiente, este problema es un caso de prueba estándar para evaluar las estrategias de sincronización.

SOLUCIÓN UTILIZANDO SEMÁFOROS

La Figura 6.12 muestra una solución utilizando semáforos. Cada filósofo toma primero el tenedor de la izquierda y después el de la derecha. Una vez que el filósofo ha terminado de comer, vuelve a colocar los dos tenedores en la mesa. Esta solución, desgraciadamente, conduce al interbloqueo: Si todos los filósofos están hambrientos al mismo tiempo, todos ellos se sentarán, asirán el tenedor de la izquierda y tenderán la mano para tomar el otro tenedor, que no estará allí. En esta indecorosa posición, los filósofos pasarán hambre.

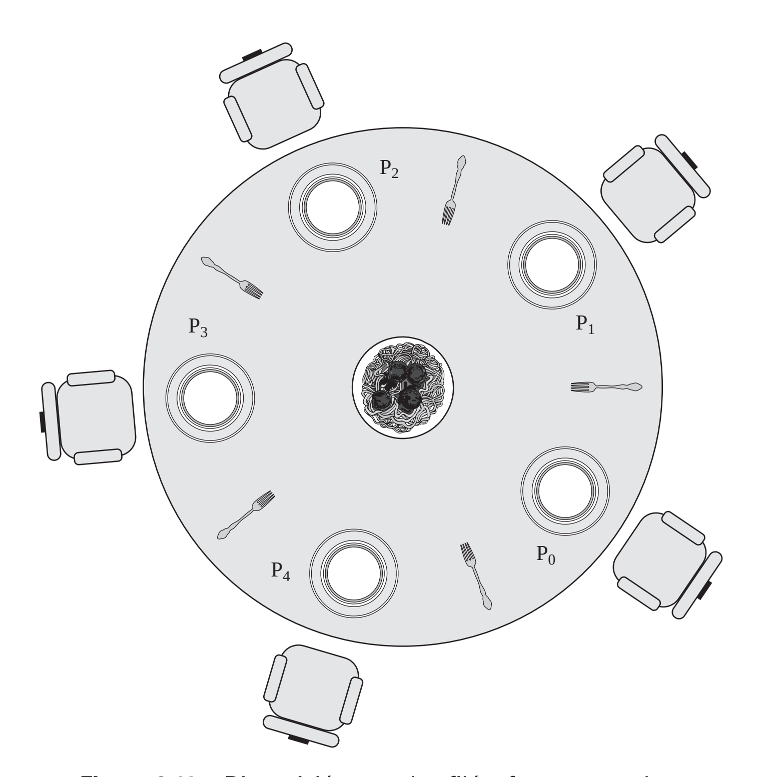


Figura 6.11. Disposición para los filósofos comensales.

Para superar el riesgo de interbloqueo, se podrían comprar cinco tenedores adicionales (una solución más higiénica) o enseñar a los filósofos a comer espaguetis con un solo tenedor. Como alternativa, se podría incorporar un asistente que sólo permitiera que haya cuatro filósofos al mismo tiempo en el comedor. Con un máximo de cuatro filósofos sentados, al menos un filósofo tendrá acceso a los dos tenedores. La Figura 6.13 muestra esta solución, utilizando de nuevo semáforos. Esta solución está libre de interbloqueos e inanición.

SOLUCIÓN UTILIZANDO UN MONITOR

La Figura 6.14 muestra una solución al problema de los filósofos comensales utilizando un monitor. Se define un vector de cinco variables de condición, una variable de condición por cada tenedor. Estas variables de condición se utilizan para permitir que un filósofo espere hasta que esté disponible un tenedor. Además, hay un vector de tipo booleano que registra la disponibilidad de cada tenedor (`verdadero` significa que el tenedor está disponible). El monitor consta de dos procedimientos. El procedimiento `obtiene_tenedores` lo utiliza un filósofo para obtener sus tenedores, el situado a su izquierda y a su derecha. Si ambos tenedores están disponibles, el proceso que corresponde con el filósofo se encola en la variable de condición correspondiente. Esto permite que otro proceso filósofo entre en el monitor. Se utiliza el procedimiento `libera_tenedores` para hacer que queden disponibles los dos tenedores. Nótese que la estructura de esta solución es similar a la solución del semáforo propuesta en la Figura 6.12. En ambos casos, un filósofo toma primero el tenedor de la izquierda y después el de la derecha. A diferencia de la solución del semáforo, esta solución del monitor no sufre interbloqueos, porque sólo puede haber un proceso en cada momento en el monitor. Por ejemplo, se

```
/* programa filosofos_comensales */
semaforo tenedor [5] = {1};
int i;
void filosofo (int i)
{
    while (verdadero)
    {
        piensa ();
        wait (tenedor[i]);
        wait (tenedor[(i+1) mod 5]);
        come ();
        signal (tenedor[(i+1) mod 5]);
        signal (tenedor[i]);
    }
}
void main ()
{
    paralelos (filosofo (0), filosofo (1), filosofo (2),
 filosofo (3), filosofo (4));
}
```

Figura 6.12. Una primera solución al problema de los filósofos comensales.

```
/* programa filosofos_comensales */
semaforo tenedor [5] = {1};
semaforo comedor = {4};
int i;
void filosofo (int i)
{
    while (verdadero)
    {
        piensa ();
        wait (comedor);
        wait (tenedor[i]);
        wait (tenedor[(i+1) mod 5]);
        come ();
        signal (tenedor[(i+1) mod 5]);
        signal (tenedor[i]);
        signal (comedor);
    }
}
void main ()
{
    paralelos (filosofo (0), filosofo (1), filosofo (2),
 filosofo (3), filosofo (4));
}
```

Figura 6.13. Una segunda solución al problema de los filósofos comensales.

garantiza que el primer proceso filósofo que entre en el monitor pueda asir el tenedor de la derecha después de que tome el de la izquierda pero antes de que el siguiente filósofo a la derecha tenga una oportunidad de asir el tenedor de su izquierda, que es el que está a la derecha de este filósofo.

6.7. MECANISMOS DE CONCURRENCIA DE UNIX

UNIX proporciona diversos mecanismos de comunicación y sincronización entre procesos. En esta sección, se revisarán los más importantes:

- Tuberías (*pipes*).
- Mensajes.
- Memoria compartida.
- Semáforos.
- Señales.

Las tuberías, los mensajes y la memoria compartida pueden utilizarse para comunicar datos entre procesos, mientras que los semáforos y las señales se utilizan para disparar acciones en otros procesos.

```
monitor controlador_de_comensales;
cond TenedorListo[5];                    /* variable de condición para sincronizar */
boolean tenedor [5] = {verdadero};             /* disponibilidad de cada tenedor */
void obtiene_tenedores (int id_pr)    /* id_pr es el número de ident. del filósofo */
{
    int izquierdo = id_pr;
    int derecho = (id_pr++) % 5;
    /* concede el tenedor izquierdo */
    if (!tenedor(izquierdo))
        cwait(TenedorListo[izquierdo]);          /* encola en variable de condición */
    tenedor(izquierdo) = falso;
    /* concede el tenedor derecho */
    if (!tenedor(derecho))
        cwait(TenedorListo[derecho]);            /* encola en variable de condición */
    tenedor(derecho) = falso;
}
void libera_tenedores (int id_pr)
{
    int izquierdo = id_pr;
    int derecho = (id_pr++) % 5;
    /* libera el tenedor izquierdo */
    if (empty(TenedorListo[izquierdo]))             /* nadie espera por este tenedor */
        tenedor(izquierdo) = verdadero;
    else   /* despierta a un proceso que espera por este tenedor */
        csignal(TenedorListo[izquierdo]);
    /*libera el tenedor derecho*/
    if (empty(TenedorListo[derecho]))               /* nadie espera por este tenedor */
        tenedor(derecho) = verdadero;
    else                      /* despierta a un proceso que espera por este tenedor */
        csignal(TenedorListo[derecho]);
}
```

```
void filosofo[k=0 hasta 4]                           /* los cinco clientes filósofos */
{
    while (verdadero)
    {
        <piensa>;
        obtiene_tenedores(k);         /* cliente solicita dos tenedores vía el monitor */
        <come espaguetis>;
        libera_tenedores(k);               /* cliente libera tenedores vía el monitor */
    }
}
```

Figura 6.14. Una solución al problema de los filósofos comensales usando un monitor.

TUBERÍAS

Una de las contribuciones más significativas de UNIX en el desarrollo de los sistemas operativos es la tubería. Inspirado por el concepto de corrutina [RITC84], una tubería es un *buffer* circular que permite que dos procesos se comuniquen siguiendo el modelo productor-consumidor. Por tanto, se trata de una cola de tipo el primero en entrar es el primero en salir, en la que escribe un proceso y lee otro.

Cuando se crea una tubería, se le establece un tamaño fijo en bytes. Cuando un proceso intenta escribir en la tubería, la petición de escritura se ejecuta inmediatamente si hay suficiente espacio; en caso contrario, el proceso se bloquea. De manera similar, un proceso que lee se bloquea si intenta leer más bytes de los que están actualmente en la tubería; en caso contrario, la petición de lectura se ejecuta inmediatamente. El sistema operativo asegura la exclusión mutua: es decir, en cada momento sólo puede acceder a una tubería un único proceso.

Hay dos tipos de tuberías: con nombre y sin nombre. Sólo los procesos relacionados pueden compartir tuberías sin nombre, mientras que, los procesos pueden compartir tuberías con nombre tanto si están relacionados como si no.

MENSAJES

Un mensaje es un conjunto de bytes con un tipo asociado. UNIX proporciona las llamadas al sistema `msgsnd` y `msgrcv` para que los procesos puedan realizar la transferencia de mensajes. Asociada con cada proceso existe una cola de mensajes, que funciona como un buzón.

El emisor del mensaje especifica el tipo de mensaje en cada mensaje que envía, de manera que este tipo puede utilizarse como un criterio de selección por el receptor. El receptor puede recuperar los mensajes tanto en orden de llegada o por su tipo. Un proceso se bloqueará cuando intente enviar un mensaje a una cola llena. Un proceso también se bloqueará cuando intente leer un mensaje de una cola vacía. Si un proceso intenta leer un mensaje de un cierto tipo y no es posible debido a que no está presente ningún mensaje de este tipo, el proceso no se bloquea.

MEMORIA COMPARTIDA

La forma más rápida de comunicación entre procesos proporcionada en UNIX es la memoria compartida. Se trata de un bloque de memoria virtual compartido por múltiples procesos. Los procesos leen y escriben en la memoria compartida utilizando las mismas instrucciones de máquina que se utilizan para leer y escribir otras partes de su espacio de memoria virtual. El permiso para un determinado proceso puede ser de sólo lectura o de lectura y escritura, estableciéndose de forma individual para cada proceso. Las restricciones de exclusión mutua no son parte de este mecanismo de memoria compartida sino que las deben proporcionar los procesos que utilizan la memoria compartida.

SEMÁFOROS

Las llamadas al sistema de semáforos en UNIX System V son una generalización de las funciones `semWait` y `semSignal` definidas en el Capítulo 5; se pueden realizar varias operaciones simultáneamente y las operaciones de incremento y decremento pueden corresponder con valores mayores que 1. El núcleo realiza todas las operaciones solicitadas atómicamente; ningún otro proceso puede acceder al semáforo hasta que se hayan completado todas las operaciones.

Un semáforo consta de los siguientes elementos:

- El valor actual del semáforo.
- El identificador del último proceso que operó con el semáforo.
- El número de procesos en espera de que el valor del semáforo sea mayor que su valor actual.
- El número de procesos en espera de que el valor del semáforo sea cero.
- Asociado con el semáforo están las colas de los procesos bloqueados en ese semáforo.

Los semáforos se crean realmente como conjuntos, constando cada conjunto de semáforos de uno o más semáforos. Hay una llamada al sistema `semctl` que permite que todos los valores de los semáforos del conjunto se fijen al mismo tiempo. Además, hay una llamada al sistema `semop` que toma como argumento una lista de operaciones de semáforo, cada una definida sobre uno de los semáforos de un conjunto. Cuando se realiza esta llamada, el núcleo lleva a cabo sucesivamente las operaciones indicadas. Por cada operación, la función real se especifica mediante el valor `sem_op,` existiendo las siguientes posibilidades:

- Si `sem_op` es positivo, el núcleo incrementa el valor del semáforo y despierta a todos los procesos en espera de que el valor del semáforo se incremente.
- Si `sem_op` es 0, el núcleo comprueba el valor del semáforo. Si el valor del semáforo es igual a 0, el núcleo continúa con las otras operaciones de la lista. En caso contrario, el núcleo incrementa el número de procesos en espera de que ese semáforo tenga el valor 0 y suspende al proceso para que espere por el evento de que el valor del semáforo se haga igual a 0.
- Si `sem_op` es negativo y su valor absoluto es menor o igual al valor sel semáforo, el núcleo añade `sem_op` (un número negativo) al valor del semáforo. Si el resultado es 0, el núcleo despierta a todos los procesos en espera de que el semáforo tome ese valor.
- Si `sem_op` es negativo y su valor absoluto es mayor que el valor del semáforo, el núcleo suspende al proceso en espera del evento de que el valor del semáforo se incremente.

Esta generalización de los semáforos proporciona una considerable flexibilidad para realizar la sincronización y coordinación de procesos.

SEÑALES

Una señal es un mecanismo software que informa a un proceso de la existencia de eventos asíncronos. Una señal es similar a las interrupciones hardware pero no emplea prioridades. Es decir, todas las señales se tratan por igual; las señales que ocurren al mismo tiempo se le presentan al proceso una detrás de otra, sin ningún orden en particular.

Los procesos se pueden enviar señales entre sí, o puede ser el núcleo quien envíe señales internamente. Para entregar una señal, se actualiza un campo en la tabla de procesos correspondiente al proceso al que se le está enviando la señal. Dado que por cada señal se mantiene un único bit, las señales de un determinado tipo no pueden encolarse. Una señal sólo se procesa después de que un proceso se despierte para ejecutar o cuando el proceso está retornando de una llamada al sistema. Un proceso puede responder a una señal realizando alguna acción por defecto (por ejemplo, su terminación), ejecutando una función de manejo de la señal o ignorando la señal.

La Tabla 6.2 enumera las señales definidas por UNIX SVR4.

Tabla 6.2. Señales UNIX.

Valor	Nombre	Descripción
01	SIGHUP	Desconexión; enviada al proceso cuando el núcleo asume que el usuario de ese proceso no está haciendo trabajo útil
02	SIGINT	Interrupción
03	SIGQUIT	Abandonar; enviada por el usuario para provocar la parada del proceso y la generación de un volcado de su memoria (*core dump*)
04	SIGILL	Instrucción ilegal
05	SIGTRAP	*Trap* de traza; activa la ejecución de código para realizar un seguimiento de la ejecución del proceso
06	SIGIOT	Instrucción IOT
07	SIGEMT	Instrucción EMT
08	SIGFPE	Excepción de coma flotante
09	SIGKILL	Matar; terminar el proceso
10	SIGBUS	Error de bus
11	SIGSEGV	Violación de segmento; el proceso intenta acceder a una posición fuera de su espacio de direcciones
12	SIGSYS	Argumento erróneo en una llamada al sistema
13	SIGPIPE	Escritura sobre una tubería que no tiene lectores asociados
14	SIGALRM	Alarma; emitida cuando un proceso desea recibir una señal cuando transcurra un determinado periodo de tiempo
15	SIGTERM	Terminación por software
16	SIGUSR1	Señal 1 definida por el usuario
17	SIGUSR2	Señal 2 definida por el usuario
18	SIGCHLD	Muerte de un proceso hijo
19	SIGPWR	Interrupción en el suministro de energía

6.8. MECANISMOS DE CONCURRENCIA DEL NÚCLEO DE LINUX

Linux incluye todos los mecanismos de concurrencia presentes en otros sistemas UNIX, como SVR4, incluyendo tuberías, mensajes, memoria compartida y señales. Además, Linux 2.6 incluye un abundante conjunto de mecanismos de concurrencia especialmente destinados para su uso cuando se está ejecutando un hilo en modo núcleo. Es decir, se trata de mecanismos utilizados dentro del núcleo para proporcionar concurrencia en la ejecución del código de núcleo. Esta sección estudiará los mecanismos de concurrencia del núcleo Linux.

OPERACIONES ATÓMICAS

Linux proporciona un conjunto de funciones que garantiza que las operaciones sobre una variable sean atómicas. Estas operaciones pueden utilizarse para evitar condiciones de carrera sencillas. Una operación atómica ejecuta sin interrupción y sin interferencia. En un sistema uniprocesador, un hilo que realiza una operación atómica no puede verse interrumpido una vez que ha comenzado la opera-

ción hasta que ésta termina. Además, en un sistema multiprocesador, se establece un cerrojo sobre la variable con la que se está operando para impedir el acceso de otros hilos hasta que se complete esta operación.

En Linux se definen dos tipos de operaciones atómicas: operaciones con enteros, que operan con una variable de tipo entero, y operaciones con mapas de bits, que operan con un bit de un mapa de bits (Tabla 6.3). Estas operaciones deben implementarse en cualquier arquitectura sobre la que se pretenda que ejecute Linux. Para algunas arquitecturas, hay instrucciones en lenguaje ensamblador que corresponden directamente con estas operaciones atómicas. En otras arquitecturas, se utiliza una operación que establece un cerrojo en el bus de memoria para garantizar que la operación sea atómica.

Tabla 6.3. Operaciones atómicas en Linux.

Operaciones atómicas con enteros	
ATOMIC_INT (int i)	En una declaración: inicia un atomic_t con el valor i
int atomic_read(atomic_t *v)	Lee el valor entero de v
void atomic_set(atomic_t *v, int i)	Asigna el valor entero i a v
void atomic_add(int i, atomic_t *v)	Suma i a v
void atomic_sub(int i, atomic_t *v)	Resta i de v
void atomic_inc(atomic_t *v)	Suma 1 a v
void atomic_dec(atomic_t *v)	Resta 1 de v
int atomic_sub_and_test(int i, atomic_t *v)	Resta i de v; devuelve 1 si el resultado es cero; y 0 en caso contrario
int atomic_dec_and_test(atomic_t *v)	Resta 1 de v; devuelve 1 si el resultado es cero; y 0 en caso contrario
Operaciones atómicas con mapas de bits	
void set_bit(int n, void *dir)	Pone a 1 el bit n del mapa de bits apuntado por dir
void clear_bit(int n, void *dir)	Pone a 0 el bit n del mapa de bits apuntado por dir
void change_bit(int n, void *dir)	Invierte el valor del bit n del mapa de bits apuntado por dir
int test_and_set_bit(int n, void *dir)	Pone a 1 el bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_and_clear_bit(int n, void *dir)	Pone a 0 el bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_and_change_bit(int n, void *dir)	Invierte el valor del bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_bit(int n, void *dir)	Devuelve el valor del bit n del mapa de bits apuntado por dir

Para las **operaciones atómicas con enteros**, se utiliza un tipo de datos especial, `atomic_t`. Las operaciones atómicas con enteros pueden utilizar solamente este tipo de datos, no permitiéndose ninguna otra operación sobre el mismo. [LOVE04] enumera las siguientes ventajas de estas restricciones:

1. Las operaciones atómicas nunca se utilizan con variables que podrían en algunas circunstancias estar desprotegidas de condiciones de carrera.
2. Las variables de este tipo de datos están protegidas del uso inapropiado mediante operaciones no atómicas.
3. El compilador no puede optimizar erróneamente el acceso al valor (por ejemplo, utilizando un alias en vez de la dirección de memoria correcta).
4. Este tipo de datos sirve para esconder las diferencias específicas de cada arquitectura en su implementación.

Un uso habitual de los tipos de datos enteros atómicos es la implementación de contadores.

Las **operaciones atómicas con mapas de bits** operan sobre una secuencia de bits almacenada en una posición de memoria indicada por una variable de tipo puntero. Así, no hay un equivalente al tipo de datos `atomic_t` requerido por las operaciones atómicas con enteros.

Las operaciones atómicas son el mecanismo más simple de sincronización del núcleo. A partir de éstas, se pueden construir mecanismos de cerrojo más complejos.

CERROJOS CÍCLICOS

La técnica más frecuentemente utilizada para proteger una sección crítica en Linux es el cerrojo cíclico (*spinlock*). En un determinado momento, sólo un único hilo puede adquirir un cerrojo cíclico. Cualquier otro hilo que intente adquirir el mismo cerrojo seguirá intentándolo (de forma cíclica) hasta que pueda adquirirlo. Esencialmente, un cerrojo cíclico se construye usando una posición en memoria que contiene un valor entero que cada hilo comprueba antes de entrar en su sección crítica. Si el valor es 0, el hilo le asigna un valor igual a 1 y entra en su sección crítica. Si el valor no es igual a cero, el hilo comprueba continuamente el valor hasta que sea cero. El cerrojo cíclico es fácil de implementar pero tiene la desventaja de que los hilos que se han quedado fuera de la sección crítica continúan ejecutando en un modo con espera activa. Por tanto, los cerrojos cíclicos son más efectivos en situaciones donde el tiempo en espera hasta adquirir el cerrojo se prevé que va a ser muy breve, de un orden de magnitud inferior al correspondiente a dos cambios de contexto.

La forma de uso básica de un cerrojo cíclico es la siguiente:

```
spin_lock(&cerrojo)
/* sección crítica */
spin_unlock(&cerrojo)
```

CERROJOS CÍCLICOS BÁSICOS

El cerrojo cíclico básico (en contraposición al cerrojo cíclico de escritura y lectura explicado posteriormente) se presenta en cuatro modalidades (Tabla 6.4):

- **Sencillo.** Si el código de la sección crítica no se ejecuta desde manejadores de interrupciones o si las interrupciones están inhabilitadas durante la ejecución de la sección crítica, se puede utilizar un cerrojo cíclico simple. No afecta al estado de las interrupciones en el procesador en el que se ejecuta.

- **_irq.** Si las interrupciones están siempre inhabilitadas, se debería utilizar este tipo de cerrojo cíclico.
- **_irqsave.** Si no se conoce si las interrupciones estarán habilitadas o inhabilitadas en el momento de la ejecución, se debería utilizar esta versión. Cuando se adquiere un cerrojo, se salva el estado actual de las interrupciones del procesador local, que se restaurará cuando se libere el cerrojo.
- **_bh.** Cuando ocurre una interrupción, el manejador de interrupciones correspondiente realiza la cantidad mínima de trabajo necesaria. Un fragmento de código, llamado mitad inferior (*bottom half*), realiza el resto del trabajo asociado a la interrupción, permitiendo que la interrupción actual se habilite lo antes posible. El cerrojo cíclico `_bh` se usa para inhabilitar, y después habilitar, la ejecución de rutinas de tipo mitad inferior para evitar conflictos con la sección crítica protegida.

Tabla 6.4. Cerrojos cíclicos en Linux.

void spin_lock(spinlock_t *cerrojo)	Adquiere el cerrojo especificado, comprobando cíclicamente el valor hasta que esté disponible
void spin_lock_irq(spinlock_t *cerrojo)	Igual que spin_lock, pero prohibiendo las interrupciones en el procesador local
void spin_lock_irqsave(spinlock_t *cerrojo, unsigned long indicadores)	Igual que spin_lock_irq, pero guardando el estado actual de las interrupciones en indicadores
void spin_lock_bh(spinlock_t *cerrojo)	Igual que spin_lock, pero prohibiendo la ejecución de mitades inferiores
void spin_unlock(spinlock_t *cerrojo)	Libera el cerrojo especificado
void spin_unlock_irq(spinlock_t *cerrojo)	Libera el cerrojo especificado y habilita las interrupciones locales
void spin_unlock_irqrestore(spinlock_t *cerrojo, unsigned long indicadores)	Libera el cerrojo especificado y restaura el estado de las interrupciones locales
void spin_unlock_bh(spinlock_t *cerrojo)	Libera el cerrojo especificado y habilita la ejecución de mitades inferiores
void spin_lock_init(spinlock_t *cerrojo)	Inicia el cerrojo especificado
int spin_trylock(spinlock_t *cerrojo)	Intenta adquirir el cerrojo especificado, devolviendo un valor distinto de cero si lo posee otro proceso y cero en caso contrario
int spin_is_locked(spinlock_t *cerrojo)	Devuelve un valor distinto de cero si el cerrojo lo posee otro proceso y cero en caso contrario

El cerrojo cíclico simple se utiliza si el programador sabe que los datos protegidos no van a accederse desde un manejador de interrupción o una rutina de tipo mitad inferior. En caso contrario, se usa el tipo de cerrojo cíclico apropiado.

Los cerrojos cíclicos se implementan de una manera diferente en un sistema uniprocesador frente a una máquina multiprocesadora. En un uniprocesador, se tendrán en cuenta las siguientes consideraciones. Si se inhabilita la opción de expulsión en el núcleo, de manera que un hilo en modo núcleo no puede verse interrumpido, se eliminan los cerrojos cíclicos en tiempo de compilación, ya que no son necesarios. Si se activa la expulsión en el núcleo, lo que permite las interrupciones, también se eliminan los cerrojos cíclicos (es decir, no se realiza ninguna comprobación de una posición de memoria

asociada a un cerrojo cíclico), sino que se implementa simplemente como un código que habilita/deshabilita las interrupciones. En un sistema multiprocesador, el cerrojo cíclico se compila como código que realiza realmente la comprobación de la posición de memoria asociada al cerrojo cíclico. La utilización del mecanismo del cerrojo cíclico en el programa permite que sea independiente de si se ejecuta en un sistema uniprocesador o multiprocesador.

CERROJO CÍCLICO DE LECTURA-ESCRITURA

El cerrojo cíclico de lectura-escritura es un mecanismo que permite un mayor grado de concurrencia dentro del núcleo que el cerrojo cíclico básico. El cerrojo cíclico de lectura-escritura permite que múltiples hilos tengan acceso simultáneo a la misma estructura de datos si sólo pretenden un acceso de lectura, pero da acceso exclusivo al cerrojo cíclico si un hilo desea actualizar la estructura de datos. Cada cerrojo cíclico de lectura-escritura consta de un contador de lectores de 24 bits y un indicador de desbloqueo, que tienen la siguiente interpretación:

Contador	Indicador	Interpretación
0	1	El cerrojo cíclico está disponible para su uso
0	0	El cerrojo cíclico se ha adquirido para la escritura de 1 hilo
n ($n > 0$)	0	El cerrojo cíclico se ha adquirido para la lectura de *n* hilos
n ($n > 0$)	1	Inválido

Como ocurre con el cerrojo cíclico básico, el cerrojo cíclico de lectura-escritura tiene versiones simples, de tipo `_irq` y de tipo `_irqsave`.

Nótese que el cerrojo cíclico de lectura-escritura favorece a los lectores frente a los escritores. Si los lectores mantienen el cerrojo cíclico, mientras que haya al menos un lector, un escritor no puede obtener el cerrojo cíclico. Además, se pueden incorporar nuevos lectores al cerrojo cíclico incluso aunque haya un escritor esperando.

SEMÁFOROS

En el nivel de usuario, Linux proporciona una interfaz de semáforos que corresponde a la de UNIX SVR4. Internamente, proporciona una implementación de semáforos para su propio uso. Es decir, el código que forma parte del núcleo puede invocar a los semáforos del núcleo, pero estos no pueden accederse directamente desde un programa de usuario mediante llamadas al sistema. Se implementan como funciones dentro del núcleo y son así más eficientes que los semáforos visibles para el usuario.

Linux proporciona tres tipos de mecanismos de semáforos en el núcleo: semáforos binarios, semáforos con contador y semáforos de lectura-escritura.

SEMÁFOROS BINARIOS Y CON CONTADOR

Los semáforos binarios y con contador definidos en Linux 2.6 (Tabla 6.5) tienen la misma funcionalidad que la descrita para ambos tipos de semáforos en el Capítulo 5. Se utilizan los nombres de fun-

ción down y up para las funciones a las que se denominó semWait y semSignal en el Capítulo 5, respectivamente.

Un semáforo con contador se inicia utilizando la función sema_init, que da al semáforo un nombre y le asigna un valor inicial. Los semáforos binarios, llamados *MUTEX* en Linux, se inician mediante las funciones init_MUTEX e init_MUTEX_LOCKED, que inician el semáforo a 1 o 0, respectivamente.

Linux proporciona tres versiones de la operación down (semWait).

1. La función down corresponde a la operación tradicional semWait. Es decir, el hilo comprueba el valor del semáforo y se bloquea si no está disponible. El hilo se despertará cuando se produzca la correspondiente operación up en este semáforo. Obsérvese que se usa el mismo nombre de función tanto para semáforos con contador como para binarios.
2. La función down_interruptible permite que el hilo reciba y responda a una señal del núcleo mientras está bloqueado en la operación *down*. Si una señal despierta al hilo, la función down_interruptible incrementa el valor del contador del semáforo y devuelve un código de error conocido en Linux como -EINTR. Este valor le indica al hilo que la función del semáforo se ha abortado. En efecto, el hilo se ha visto forzado a «abandonar» el semáforo. Esta característica es útil para manejadores de dispositivos y otros servicios en los que es conveniente abortar una operación del semáforo.
3. La función down_trylock permite intentar adquirir un semáforo sin ser bloqueado. Si el semáforo está disponible, se adquiere. En caso contrario, esta función devuelve un valor distinto de cero sin bloquear el hilo.

SEMÁFOROS DE LECTURA-ESCRITURA

El semáforo de lectura-escritura clasifica a los usuarios en lectores y escritores, permitiendo múltiples lectores concurrentes (sin escritores) pero sólo un único escritor (sin lectores). En realidad, el semáforo funciona como un semáforo con contador para los lectores pero como un semáforo binario (MUTEX) para los escritores. La Tabla 6.5 muestra las operaciones básicas de los semáforos de lectura-escritura. Los semáforos de lectura-escritura utilizan un bloqueo ininterrumpible, por lo que sólo hay una versión de cada una de las operaciones *down*.

BARRERAS

En algunas arquitecturas, los compiladores o el hardware del procesador pueden cambiar el orden de los accesos a memoria del código fuente para optimizar el rendimiento. Estos cambios de orden se llevan a cabo para optimizar el uso del *pipeline* de instrucciones del procesador. Los algoritmos que los realizan contienen comprobaciones que aseguran que no se violan las dependencias de datos. Por ejemplo, en el siguiente código:

```
a = 1;
b = 1;
```

se puede modificar el orden de manera que la posición de memoria b se actualice antes de que lo haga la posición de memoria a. Sin embargo, en el siguiente código:

```
a = 1;
b = a;
```

Tabla 6.5. Semáforos en Linux.

Semáforos tradicionales	
void sema_init(struct semaphore *sem, int cont)	Inicia el semáforo creado dinámicamente con el valor cont
void init_MUTEX(struct semaphore *sem)	Inicia el semáforo creado dinámicamente con el valor 1 (inicialmente abierto)
void init_MUTEX_LOCKED(struct semaphore *sem)	Inicia el semáforo creado dinámicamente con el valor 0 (inicialmente cerrado)
void down(struct semaphore *sem)	Intenta adquirir el semáforo especificado, entrando en un bloqueo ininterrumpible si el semáforo no está disponible
int down_interruptible(struct semaphore *sem)	Intenta adquirir el semáforo especificado, entrando en un bloqueo interrumpible si el semáforo no está disponible; devuelve el valor –EINTR si se recibe una señal
int down_trylock(struct semaphore *sem)	Intenta adquirir el semáforo especificado, y devuelve un valor distinto de cero si el semáforo no está disponible
void up(struct semaphore *sem)	Libera el semáforo especificado
Semáforos de lectura-escritura	
void init_rwsem(struct rw_semaphore, *sem_le)	Inicia el semáforo creado dinámicamente con el valor 1
void down_read(struct rw_semaphore, *sem_le)	Operación down de los lectores
void up_read(struct rw_semaphore, *sem_le)	Operación up de los lectores
void down_write(struct rw_semaphore, *sem_le)	Operación down de los escritores
void up_write(struct rw_semaphore, *sem_le)	Operación up de los escritores

no se puede cambiar el orden de los accesos a memoria. Sin embargo, hay ocasiones en las que es importante que las lecturas y las escrituras se ejecuten en el orden especificado debido al uso que se hace de la información desde otro hilo o desde un dispositivo hardware.

Para fijar el orden en el que se ejecuta cada instrucción, Linux proporciona el mecanismo de la barrera de memoria. La Tabla 6.6 enumera las funciones más importantes que se definen para este mecanismo. La operación `rmb()` asegura que no se produce ninguna lectura después de la barrera definida por el lugar que ocupa `rmb()` en el código. Igualmente, la operación `wmb()` asegura que no se produce ninguna escritura después de la barrera definida por el lugar que ocupa `wmb()` en el código. La operación `mb()` proporciona una barrera tanto de lectura como de escritura.

Tabla 6.6. Operaciones de barrera de memoria en Linux.

rmb()	Impide que se cambie el orden de las lecturas evitando que se realicen después de la barrera
wmb()	Impide que se cambie el orden de las escrituras evitando que se realicen después de la barrera
mb()	Impide que se cambie el orden de las lecturas y escrituras evitando que se realicen después de la barrera
barrier()	Impide al compilador cambiar el orden de las lecturas y escrituras evitando que se realicen después de la barrera
smp_rmb()	En SMP proporciona un `rmb()` y en UP proporciona un `barrier()`
smp_wmb()	En SMP proporciona un `wmb()` y en UP proporciona un `barrier()`
smp_mb()	En SMP proporciona un `mb()` y en UP proporciona un `barrier()`

SMP = multiprocesador simétrico

UP = uniprocesador

Hay dos aspectos importantes sobre las operaciones de barrera que conviene resaltar:

1. Las barreras están relacionadas con instrucciones de máquina, concretamente, con las instrucciones de carga y almacenamiento. Por tanto, la instrucción de lenguaje de alto nivel `a = b` implica tanto una carga (lectura) de la posición `b` y un almacenamiento (escritura) en la posición `a`.
2. Las operaciones `rmb`, `wmb` y `mb` dictan el comportamiento tanto del compilador como del procesador. En el caso del compilador, la operación de barrera dicta que el compilador no cambie el orden de las instrucciones durante el proceso de compilación. En el caso del procesador, la operación de barrera dicta que cualquier instrucción pendiente de ejecución que esté incluida en el *pipeline* antes de la barrera debe completarse antes de ejecutar cualquier instrucción que se encuentre después de la barrera.

La operación `barrier()` es una operación más ligera que la operación `mb()`, puesto que sólo controla el comportamiento del compilador. Esto sería útil si se conoce que el procesador no realizará cambios de orden indeseables. Por ejemplo, los procesadores x86 no modifican el orden de las escrituras.

Las operaciones `smp_rmb`, `smp_wmb` y `smp_mb` proporcionan una optimización para el código que puede compilarse en un uniprocesador (UP) o en un multiprocesador simétrico (SMP). Estas instrucciones se definen como barreras de memoria usuales en el caso de un SMP, pero si se trata de un UP, se interpretan como barreras sólo para el compilador. Las operaciones `smp_` son útiles en situaciones en las que las dependencias de datos de interés sólo surgirán en un contexto SMP.

6.9. FUNCIONES DE SINCRONIZACIÓN DE HILOS DE SOLARIS

Además de los mecanismos de concurrencia de UNIX SVR4, Solaris proporciona cuatro funciones de sincronización de hilos:

- Cerrojos de exclusión mutua (mutex)
- Semáforos

- Cerrojos de múltiples lectores y un único escritor (lectores/escritor)
- Variables de condición

Solaris implementa estas funciones dentro del núcleo para los hilos del núcleo, pero también se proporcionan en la biblioteca de hilos para los hilos de nivel de usuario. La Figura 6.15 muestra las estructuras de datos asociadas a estas funciones. Las funciones de iniciación de estos mecanismos rellenan algunos de los campos de datos. Una vez que se crea un objeto sincronización, básicamente, sólo se pueden realizar dos operaciones: entrar (adquirir un cerrojo) y liberar (desbloquearlo). No hay mecanismos en el núcleo o en la biblioteca de hilos que aseguren la exclusión mutua o impidan el interbloqueo. Si un hilo intenta acceder a un fragmento de datos o código que debería estar protegido pero no utiliza la función de sincronización apropiada, a pesar de ello, ese acceso se produce. Si un hilo establece un cerrojo sobre un objeto y después no lo desbloquea, el núcleo no toma ninguna acción correctiva.

Todas las primitivas de sincronización requieren la existencia de una instrucción hardware que permita que un objeto sea consultado y modificado en una operación atómica, como se vio en la Sección 5.3.

CERROJO DE EXCLUSIÓN MUTUA

Un mutex se utiliza para asegurarse que sólo un hilo en cada momento puede acceder al recurso protegido por el mutex. El hilo que obtiene el mutex debe ser el mismo que lo libera. Un hilo intenta ad-

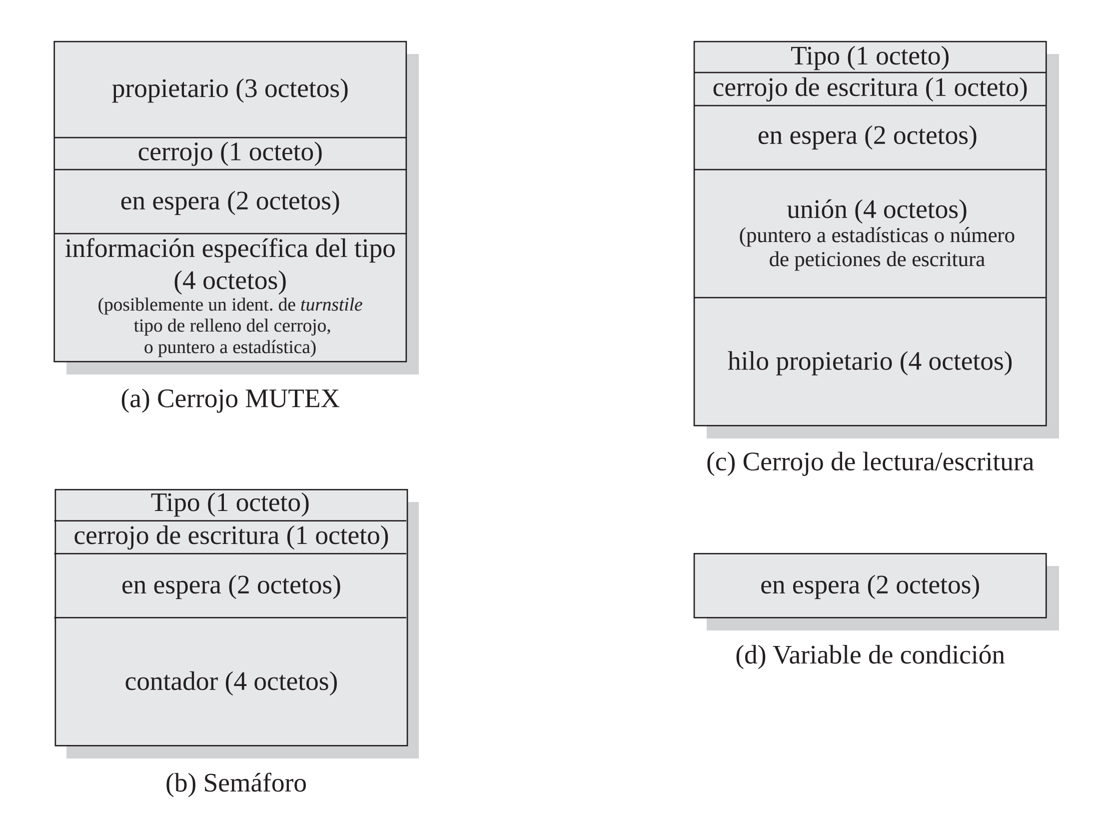


Figura 6.15. Estructura de datos de sincronización de Solaris.

quirir un cerrojo mutex ejecutando la función `mutex_enter`. Si `mutex_enter` no puede establecer el cerrojo (porque ya lo ha hecho otro hilo), la acción bloqueante depende de la información específica de tipo almacenada en el objeto mutex. La política bloqueante por defecto es la de un cerrojo cíclico: un hilo bloqueado comprueba el estado del cerrojo mientras ejecuta un bucle de espera activa. De forma opcional, hay un mecanismo de espera bloqueante basado en interrupciones. En este último caso, el mutex incluye un identificador de ***turnstile*** que identifica una cola de los hilos bloqueados en este cerrojo.

Las operaciones en un cerrojo mutex son las siguientes:

`mutex_enter()`	Adquiere el cerrojo, bloqueando potencialmente si ya lo posee otro hilo
`mutex_exit()`	Libera el cerrojo, desbloqueando potencialmente a un hilo en espera
`mutex_tryenter()`	Adquiere un cerrojo si no lo posee otro hilo

La primitiva `mutex_tryenter()` proporciona una manera no bloqueante de realizar una función de exclusión mutua. Esto permite al programador utilizar una estrategia de espera activa para hilos del nivel de usuario evitando bloquear todo el proceso cuando un hilo se bloquea.

SEMÁFOROS

Solaris proporciona los clásicos semáforos con contador, ofreciendo las siguientes funciones:

`sema_p()`	Decrementa el semáforo, bloqueando potencialmente el hilo
`sema_v()`	Incrementa el semáforo, desbloqueando potencialmente un hilo en espera
`sema_tryp()`	Decrementa el semáforo si no se requiere un bloqueo

Una vez más, la función `sema_tryp()` permite una espera activa.

CERROJO DE LECTURA/ESCRITURA

El cerrojo de lectura/escritura permite que múltiples hilos tengan acceso simultáneo de lectura a objetos protegidos por un cerrojo. También permite que en cada momento un único hilo acceda al objeto para modificarlo mientras se excluye el acceso de todos los lectores. Cuando se adquiere el cerrojo para realizar una escritura, éste toma el estado de `cerrojo de escritura`: todos los hilos que intentan acceder para leer o escribir deben esperar. Si uno o más lectores han adquirido el cerrojo, su estado es de `cerrojo de lectura`. Las funciones son las siguientes:

`rw_enter()`	Intenta adquirir un cerrojo como lector o escritor
`rw_exit()`	Libera un cerrojo como lector o escritor
`rw_tryenter()`	Decrementa el semáforo si no se requiere un bloqueo
`rw_downgrade()`	Un hilo que ha adquirido un cerrojo de escritura lo convierte en uno de lectura. Cualquier escritor en espera permanece esperando hasta que este hilo libere el cerrojo. Si no hay escritores en espera, la función despierta a los lectores pendientes, si los hay
`rw_tryupgrade()`	Intenta convertir un cerrojo de lectura en uno de escritura

VARIABLES DE CONDICIÓN

Una variable de condición se utiliza para esperar hasta que se cumpla una determinada condición. Este mecanismo debe utilizarse en conjunción con un cerrojo mutex. Con ello, se implementa un monitor del tipo mostrado en la Figura 6.14. Las primitivas son las siguientes:

`cv_wait()`	Bloquea hasta que se activa la condición
`cv_signal()`	Despierta uno de los hilos bloqueados en `cv_wait()`
`cv_broadcast()`	Despierta todos los hilos bloqueados en `cv_wait()`

La función `cv_wait()` libera el mutex asociado antes de bloquearse y lo vuelve a adquirir antes de retornar. Dado que al intentar volverlo a adquirir puede bloquearse por otros hilos que compiten por el mutex, la condición que causó la espera debe volverse a comprobar. El uso típico es el siguiente:

```
mutex_enter(&m);
• •
while (condicion) {
    cv_wait(&vc, &m);
}
• •
mutex_exit(&m);
```

Esto permite que la condición sea una expresión compleja, puesto que está protegida por el mutex.

6.10. MECANISMOS DE CONCURRENCIA DE WINDOWS

Windows XP y 2003 proporcionan sincronización entre hilos como parte de su arquitectura de objetos. Los dos métodos de sincronización más importantes son los objetos de sincronización y los de sección crítica. Los objetos de sincronización utilizan funciones de espera. Por tanto, en primer lugar, se describirán las funciones de espera y, a continuación, se examinarán los dos tipos de objetos.

FUNCIONES DE ESPERA

Las funciones de espera permiten que un hilo bloquee su propia ejecución. Las funciones de espera no retornan hasta que se cumplen los criterios especificados. El tipo de función de espera determina el conjunto de criterios utilizado. Cuando se llama a una función de espera, ésta comprueba si se satisface el criterio de espera. En caso negativo, el hilo que realizó la llamada transita al estado de espera, no usando tiempo de procesador mientras no se cumplan los criterios de la misma.

El tipo más sencillo de función de espera es aquél que espera por un solo objeto. La función `WaitForSingleObject` requiere un manejador que corresponda con un objeto de sincronización. La función retorna cuando se produce una de las siguientes circunstancias:

- El objeto especificado está en el estado de señalado.
- Ha transcurrido el plazo máximo de espera. Dicho plazo máximo puede fijarse en `INFINITE` para especificar que la espera será ilimitada.

OBJETOS DE SINCRONIZACIÓN

El mecanismo utilizado por el ejecutivo de Windows para implementar las funciones de sincronización se basa en la familia de objetos de sincronización, que se muestran con breves descripciones en la Tabla 6.7.

Tabla 6.7. Objetos de sincronización de Windows.

Tipo de objeto	Definición	Pasa al estado de señalado cuando	Efecto sobre los hilos en espera
Evento	Un aviso de que ha ocurrido un evento del sistema	Un hilo genera un evento	Desbloquea a todos
Mutex	Un mecanismo que proporciona exclusión mutua; equivalente a un semáforo binario	El hilo propietario u otro hilo libera el mutex	Se desbloquea un hilo
Semáforo	Un contador que regula el número de hilos que pueden usar un recurso	El contador del semáforo llega a cero	Desbloquea a todos
Temporizador con espera	Un contador que registra el paso del tiempo	Se cumple el tiempo especificado o expira el intervalo de tiempo	Desbloquea a todos
Notificación de cambio en fichero	Una notificación de cambios en el sistema de ficheros	Ocurre un cambio en el sistema de ficheros que encaja con los criterios de filtro de ese objeto	Se desbloquea un hilo
Entrada de consola	Una ventana de texto en la pantalla (por ejemplo, usada para manejar E/S de pantalla en una aplicación MS-DOS)	Hay entrada disponible para procesar	Se desbloquea un hilo
Trabajo	Una instancia de un fichero abierto o un dispositivo de E/S	Se completa una operación de E/S	Desbloquea a todos
Notificación sobre el recurso de memoria	Una notificación de un cambio en el recurso de memoria	Se produce el tipo de cambio especificado en la memoria física	Desbloquea a todos
Proceso	Una invocación de un programa, incluyendo el espacio de direcciones y los recursos requeridos para ejecutar el programa	El último hilo termina	Desbloquea a todos
Hilo	Una entidad ejecutable dentro de un proceso	El hilo termina	Desbloquea a todos

Nota: las filas que no están coloreadas corresponden a objetos que sólo existen para sincronización.

Los primeros cuatro tipos de objetos de la tabla están diseñados específicamente para dar soporte a la sincronización. Los tipos de objetos restantes tienen otros usos adicionales pero también pueden utilizarse para la sincronización.

Cada instancia de un objeto de sincronización puede estar en el estado de señalado o de no señalado. Un hilo se puede bloquear en un objeto si está en el estado de no señalado, desbloqueándose cuando el objeto transite al estado de señalado. El mecanismo es sencillo: un hilo realiza una petición de espera al ejecutivo de Windows, utilizando el manejador del objeto de sincronización. Cuando el objeto transita al estado de señalado, el ejecutivo de Windows desbloquea todos los objetos de tipo hilo que están esperando en ese objeto de sincronización.

El **objeto evento** es útil para enviar una señal a un hilo para indicarle que ha ocurrido un determinado evento. Por ejemplo, en la entrada o salida asíncrona, el sistema establece un objeto evento específico de manera que dicho objeto transitará al estado de señalado cuando se haya completado la operación asíncrona. El **objeto mutex** se usa para garantizar el acceso mutuamente exclusivo a un recurso, permitiendo que, en cada momento, sólo un hilo pueda conseguir el acceso al mismo. Este tipo de objeto funciona, por tanto, como un semáforo binario. Cuando el objeto mutex pasa al estado de señalado, sólo se desbloquea uno de los hilos que estaba esperando por el mutex. Los mutex se pueden utilizar para sincronizar hilos que se ejecutan en procesos diferentes. Como los mutex, los **objetos semáforo** pueden compartir los hilos pertenecientes a distintos procesos. El semáforo de Windows es un semáforo con contador. Básicamente, el **objeto temporizador con espera** avisa cuando ha transcurrido un cierto tiempo o en intervalos regulares.

OBJETOS DE SECCIÓN CRÍTICA

Los objetos de sección crítica proporcionan un mecanismo de sincronización similar al proporcionado por los objetos mutex, excepto que los objetos de sección crítica sólo los pueden utilizar hilos del mismo proceso. Los objetos mutex, eventos y semáforos se pueden utilizar también en una aplicación que tenga un único proceso, pero los objetos de sección crítica proporcionan un mecanismo de sincronización para exclusión mutua ligeramente más rápido y más eficiente.

El proceso es el responsable de asignar la memoria utilizada por una sección crítica. Normalmente, esto se hace simplemente declarando una variable de tipo `CRITICAL_SECTION`. Antes de que los hilos del proceso puedan utilizarla, la sección crítica se inicia utilizando las funciones `InitializeCriticalSection` o `InitializeCriticalSectionAndSpinCount`.

Un hilo usa las funciones `EnterCriticalSection` o `TryEnterCriticalSection` para solicitar la posesión de una sección crítica, utilizando la función `LeaveCriticalSection` para liberar la posesión de la misma. Si el objeto de sección crítica lo posee actualmente otro hilo, `EnterCriticalSection` espera indefinidamente hasta poder obtener su posesión. En contraste, cuando se utiliza un objeto mutex para lograr exclusión mutua, las funciones de espera aceptan que se especifique un plazo de tiempo de espera máximo. La función `TryEnterCriticalSection` intenta entrar en una sección crítica sin bloquear el hilo que realizó la llamada.

6.11. RESUMEN

El interbloqueo es el bloqueo de un conjunto de procesos que o bien compiten por recursos del sistema o se comunican entre sí. El bloqueo es permanente a menos que el sistema operativo tome acciones correctivas, como abortar uno o más procesos, o forzar que la ejecución de uno o más procesos retroceda. El interbloqueo puede involucrar a recursos reutilizables o consumibles. Un recurso reutilizable es aquél que no se destruye cuando se usa, como un canal de E/S o una región de memoria. Un

recurso consumible es aquél que se destruye cuando lo adquiere un proceso; algunos ejemplos son los mensajes y la información almacenada en *buffers* de E/S.

Hay tres estrategias generales para tratar con los interbloqueos: prevención, detección y predicción. La prevención del interbloqueo garantiza que no se produce el interbloqueo asegurándose de que no se cumple una de las condiciones necesarias para el interbloqueo. La detección del interbloqueo es necesaria si el sistema operativo está siempre dispuesto a conceder peticiones de recursos; periódicamente, el sistema operativo debe comprobar si hay interbloqueo y tomar las acciones pertinentes para romperlo. La predicción del interbloqueo implica el análisis de cada nueva petición de un recurso para determinar si podría conducir a un interbloqueo, concediéndola sólo si no es posible el interbloqueo.

6.12. LECTURAS RECOMENDADAS

El artículo clásico sobre interbloqueos, [HOLT72], sigue siendo una lectura valiosa, como lo es también [COFF71]. Otro estudio interesante es [ISLO80]. [CORB96] es un tratado general sobre la detección de interbloqueos. [DIMI98] es un ameno estudio general de los interbloqueos. Dos artículos recientes de Levine [LEVI03a, LEVI03b] clarifican algunos conceptos utilizados en el estudio del interbloqueo. [SHUB03] es un estudio útil del interbloqueo.

Los mecanismos de concurrencia de UNIX SVR4, Linux y Solaris 2 están detalladamente estudiados en [GRAY97], [LOVE04] y [MAUR01], respectivamente.

COFF71 Coffman, E.; Elphick, M.; y Shoshani, A. «System Deadlocks.» *Computing Surveys*, Junio 1971.

CORB96 Corbett, J. «Evaluating Deadlock Detection Methods for Concurrent Software». *IEEE Transactions on Software Engineering*, Marzo 1996.

DIMI98 Dimitoglou, G. «Deadlocks and Methods for Their Detection, Prevention, and Recovery in Modern Operating Systems.» *Operating Systems Review*, Julio 1998.

GRAY97 Gray, J. *Interprocess Communications in UNIX: The Nooks and Crannies*. Upper Saddle River, NJ: Prentice Hall, 1997.

HOLT72 Holt, R. «Some Deadlock Properties of Computer Systems.» *Computing Surveys*, Septiembre 1972.

ISLO80 Isloor, S., y Marsland, T. «The Deadlock Problem: An Overview.» *Computer*, Septiembre 1980.

LEVI03a Levine, G. «Defining Deadlock. (Operating Systems Review), Enero 2003.

LEVI03b Levine, G. «Defining Deadlock with Fungible Resources.» *Operating Systems Review*, Julio 2003.

LOVE04 Love, R. *Linux Kernel Development*. Indianapolis, IN: Sams Publishing, 2004.

MAUR01 Mauro, J., McDougall, R. *Solaris Internals*. Upper Saddle River, NJ: Prentice Hall PTR, 2001.

SHUB03 Shub, C. «A Unified Treatment of Deadlock.» *Journal of Computing in Small Colleges*, Octubre 2003. Disponible en la biblioteca digital de ACM.

6.13. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

algoritmo del banquero	exclusión mutua	predicción del interbloqueo
barrera de memoria	expropiación	prevención del interbloqueo
cerrojo cíclico	grafo de asignación de recursos	recurso consumible
detección del interbloqueo	inanición	recurso reutilizable
diagrama de progreso conjunto	interbloqueo	retención y espera
espera circular	mensaje	tubería

CUESTIONES DE REPASO

6.1. Cite ejemplos de recursos reutilizables y consumibles.

6.2. ¿Cuáles son las tres condiciones que deben cumplirse para que sea posible un interbloqueo?

6.3. ¿Cuáles son las cuatro condiciones que producen un interbloqueo?

6.4. ¿Cómo se puede prever la condición de retención y espera?

6.5. Enumere dos maneras cómo se puede prever la condición de sin expropiación.

6.6. ¿Cómo se puede prever la condición de espera circular?

6.7. ¿Cuál es la diferencia entre predicción, detección y prevención del interbloqueo?

PROBLEMAS

6.1. Muestre las cuatro condiciones del interbloqueo aplicadas a la Figura 6.1a.

6.2. Para la Figura 6.3, proporcione una descripción narrativa de cada una de las seis trayectorias representadas, similar a la descripción de la Figura 6.2 realizada en la Sección 6.1.

6.3. Se afirmó que no puede ocurrir un interbloqueo en la situación reflejada en la Figura 6.3. Justifique esa afirmación.

6.4. Considere la siguiente instantánea del sistema. Suponga que no hay peticiones de recursos pendientes de satisfacerse.

disponibles

r1	r2	r3	r4
2	1	0	0

	asignación actual				necesidades máximas				necesidades pendientes			
proceso	**r1**	**r2**	**r3**	**r4**	**r1**	**r2**	**r3**	**r4**	**r1**	**r2**	**r3**	**r4**
p1	0	0	1	2	0	0	1	2				
p2	2	0	0	0	2	7	5	0				
p3	0	0	3	4	6	6	5	6				
p4	2	3	5	4	4	3	5	6				
p5	0	3	3	2	0	6	5	2				

a) Calcule cuánto más podría pedir todavía cada proceso y escríbalo en las columnas etiquetadas como «necesidades pendientes».

b) ¿Está el sistema actualmente en un estado seguro o inseguro? ¿Por qué?

c) ¿Está este sistema actualmente en un interbloqueo? ¿Por qué o por qué no?

d) ¿Qué procesos, en caso de que haya alguno, están o pueden llegar a estar en interbloqueo?

e) Si llega una solicitud de p3 de (0, 1, 0, 0), ¿puede concederse esa solicitud inmediatamente de forma segura? ¿En qué estado (interbloqueo, seguro o inseguro) quedaría el sistema justo después de la concesión de esa petición completa? ¿Qué procesos, en caso de que haya alguno, estarían en interbloqueo si se concede inmediatamente esa petición completa?

6.5. Aplique el algoritmo de detección del interbloqueo de la Sección 6.4 a los siguientes datos y muestre el resultado.

$$\text{Disponibles} = (2\ \ 1\ \ 0\ \ 0)$$

$$\text{Solicitud} = \begin{vmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{vmatrix} \qquad \text{Asignación} = \begin{vmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{vmatrix}$$

6.6. Un sistema de *spooling* (Figura 6.16) consta de un proceso de entrada E, un proceso usuario P, y un proceso de salida S, conectados por dos *buffers* residentes en un disco. Los procesos intercambian datos en bloques de igual tamaño. Estos bloques se almacenan en el disco utilizando un límite flotante entre los *buffers* de entrada y de salida, dependiendo de la velocidad de los procesos. Las funciones de comunicación utilizadas aseguran que se cumple la siguiente restricción con respecto a los recursos:

$$e + s \leq máx$$

donde

máx = máximo número de bloques en disco

e = número de bloques de entrada en disco

s = número de bloques de salida en disco

Con respecto a los procesos, se conocen los siguientes aspectos:

1. Mientras que el entorno proporcione datos, el proceso E, en un momento dado, los escribirá en el disco (siempre que haya espacio disponible en el mismo).
2. Mientras que haya entrada disponible en el disco, el proceso P, en un momento dado, la consumirá y escribirá en el disco una cantidad finita de datos de salida por cada bloque leído (siempre que haya espacio disponible en el mismo).
3. Mientras que haya datos de salida disponibles en el disco, el proceso S, en un momento dado, los consumirá. Demuestre que en este sistema puede producirse un interbloqueo.

6.7. Sugiera una restricción adicional con respecto a los recursos que impida el interbloqueo del Problema 6.6 pero que siga permitiendo que el límite entre los *buffers* de entrada y de salida pueda variar de acuerdo a las necesidades presentes de los procesos.

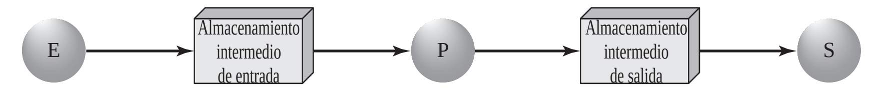


Figura 6.16. Un sistema de *spooling.*

6.8. En el sistema operativo THE [DIJK68], un tambor (precursor del disco para almacenamiento secundario) se divide en *buffers* de entrada, áreas de procesamiento y *buffers* de salida, con límites flotantes, dependiendo de la velocidad de los procesos involucrados. El estado actual del tambor se puede caracterizar por los siguientes parámetros:

máx = máximo número de páginas en el tambor

e = número de páginas de entrada en el tambor

p = número de páginas de procesamiento en el tambor

s = número de páginas de salida en el tambor

ress = número de páginas reservadas para la salida

resp = número de páginas reservadas para el procesamiento

Formule las restricciones necesarias con respecto a los recursos de manera que se garantice que no se excede la capacidad del tambor y que se reserva un número mínimo de páginas permanentemente para la salida y el procesamiento.

6.9. En el sistema operativo THE, una página puede realizar las siguientes transiciones:

1. vacío → *buffer* de entrada (producción de entrada)
2. *buffer* de entrada → área de procesamiento (consumo de entrada)
3. área de procesamiento → *buffer* de salida (producción de salida)
4. *buffer* de salida → vacío (consumo de salida)
5. vacío → área de procesamiento (llamada a procedimiento)
6. área de procesamiento → vacío (retorno de procedimiento)

a) Defina el efecto de estas transiciones en términos de las cantidades *e*, *s* y *p*.

b) ¿Puede alguna de ellas llevar a un interbloqueo si se mantienen las suposiciones hechas en el Problema 6.6 con respecto a los procesos de entrada, a los de usuario y a los de salida?

6.10. Considere un sistema con un total de 150 unidades de memoria, asignadas a tres procesos como se muestra a continuación

Proceso	**Máx**	**Asignadas**
1	70	45
2	60	40
3	60	15

Aplique el algoritmo del banquero para determinar si sería seguro conceder cada una de las siguientes peticiones. En caso afirmativo, indique una secuencia de terminaciones que se puede garantizar como factible. En caso negativo, muestre la reducción de la tabla de asignación resultante.

a) Aparece un cuarto proceso, con una necesidad de memoria máxima de 60 y una necesidad inicial de 25 unidades.

b) Aparece un cuarto proceso, con una necesidad de memoria máxima de 60 y una necesidad inicial de 35 unidades.

6.11. Evalúe el grado de utilidad del algoritmo del banquero en un sistema operativo.

6.12. Se implementa un algoritmo con una estructura en serie (*pipeline*) de manera que un flujo de elementos de datos de tipo **T** producido por un proceso P_0 pasa a través de una secuencia de procesos P_1, P_2, ..., P_{n-1}, que operan sobre los elementos en ese orden.

a) Defina un *buffer* general de mensajes que contenga todos los elementos de datos parcialmente consumidos y escriba un algoritmo para el proceso P_i ($0 \leq i \leq n-1$), con la siguiente estructura:

repetir indefinidamente
    recibe desde el predecesor;
    consume elemento;
    envía a sucesor;
por siempre

Suponga que P_0 recibe los elementos enviados por P_{n-1}. El algoritmo debería permitir que los procesos trabajen directamente con los mensajes guardados en el *buffer* de manera que no sea necesario hacer copias.

b) Muestre que los procesos no pueden quedarse en un interbloqueo con respecto al uso del *buffer* común.

6.13. a) Tres procesos comparten cuatro unidades de un recurso que pueden reservarse y liberarse sólo una en cada momento. La necesidad máxima de cada proceso es de dos unidades. Demuestre que no puede ocurrir un interbloqueo.

b) *N* procesos comparten *M* unidades de un recurso que pueden reservarse y liberarse sólo una en cada momento. La necesidad máxima de cada proceso no excede de *M*, y la suma de todas las necesidades máximas es menor que *M* + *N*. Demuestre que no puede ocurrir un interbloqueo.

6.14. Considere un sistema formado por cuatro procesos y un solo recurso. El estado actual de las matrices de asignación y necesidad es el siguiente:

$$N = \begin{vmatrix} 3 \\ 2 \\ 9 \\ 7 \end{vmatrix} \qquad A = \begin{vmatrix} 1 \\ 1 \\ 3 \\ 2 \end{vmatrix}$$

¿Cuál es el número mínimo de unidades del recurso que necesitan estar disponibles para que este estado sea seguro?

6.15. Considere los siguientes modos de manejo del interbloqueo: (1) algoritmo del banquero, (2) detección del interbloqueo y aborto del hilo, liberando todos los recursos, (3) reserva de todos los recursos por anticipado, (4) reinicio del hilo y liberación de todos los recursos si el hilo necesita esperar, (5) ordenamiento de recursos, y (6) detección de interbloqueo y retroceso en la ejecución del hilo.

a) Un criterio utilizado en la evaluación de diferentes estrategias de tratamiento del interbloqueo es analizar cuál permite mayor concurrencia. En otras palabras, ¿qué estrategia permite ejecutar sin esperas a la mayoría de los hilos cuando no hay interbloqueo? Asigne un orden de 1 a 6 a cada una de las estrategias de tratamiento del interbloqueo, tal que 1 permite el mayor grado de concurrencia. Explique su ordenación.

b) Otro criterio es la eficiencia; en otras palabras, ¿qué estrategia sobrecarga menos al procesador? Ordene las estrategias desde 1 a 6, siendo 1 la más eficiente, asumiendo que el interbloqueo es un evento que raramente se produce. Explique su ordenación. ¿Cambia su respuesta si el interbloqueo fuera frecuente?

6.16. Comente la siguiente solución al problema de los filósofos comensales. Un filósofo hambriento toma primero el tenedor de su izquierda; si también está disponible el de la derecha, lo tomará y empezará a comer; en caso contrario, devolverá el tenedor de la izquierda y repetirá el ciclo.

6.17. Supóngase que hay dos tipos de filósofos. Uno de ellos siempre toma primero el tenedor de la izquierda (un «zurdo»), y el otro tipo siempre toma primero el de su derecha (un «diestro»). El comportamiento del zurdo se define en la Figura 6.12, mientras que el del diestro es el siguiente:

```
inicio
    repetir indefinidamente
        piensa;
        wait (tenedor[(i+1) mod 5]);
        wait (tenedor[i]);
        come;
        signal (tenedor[i]);
        signal (tenedor[(i+1) mod 5]);
    porsiempre
fin
```

Demuestre lo siguiente:

a) Cualquier combinación de zurdos y diestros con al menos uno de cada evita el interbloqueo.

b) Cualquier combinación de zurdos y diestros con al menos uno de cada impide la inanición.

6.18. La Figura 6.17 muestra otra solución al problema de los filósofos comensales usando monitores. Compare esta solución con la presentada en la Figura 6.14 y escriba sus conclusiones.

6.19. En la Tabla 6.3, algunas de las operaciones atómicas de Linux no involucran dos accesos a una variable, como `atomic_read(atomic_t *v)`. Una simple operación de lectura es obviamente atómica en cualquier arquitectura. Por tanto, ¿por qué se añade esta operación al repertorio de operaciones atómicas?

6.20. Considere el siguiente fragmento de código en un sistema Linux:

```
read_lock (&cerrojo_le);
write_lock (&cerrojo_le);
```

Donde `cerrojo_le` es un cerrojo de lectura_escritura. ¿Cuál es el efecto de este código?

6.21. Las dos variables a y b tienen valores iniciales de 1 y 2 respectivamente. Dado el siguiente código que se ejecuta en un sistema Linux:

```
monitor controlador_ de_comensales;
enum estados {pensando, hambriento, comiendo} estado[5];
cond requiereTenedor [5];                                   /* variable de condición */
void obtiene_tenedores (int id_pr)          /* id_pr es el número de ident. del filósofo */
{
    estado[id_pr] = hambriento;                              /* anuncia que tiene hambre */
    if ((estado[(id_pr+1) % 5] == comiendo)
     || (estado[(id_pr-1) % 5] == comiendo))
        cwait(requiereTenedor [id_pr]);                  /* espera si algún vecino come */
    estado[id_pr] = comiendo;                         /* continúa si ningún vecino come */
}
void libera_tenedores (int id_pr)
{
    estado[id_pr] = pensando;
    /* le da una oportunidad de comer al siguiente vecino (número mayor) */
    if ((estado[(id_pr+1) % 5] == hambriento)
     && (estado[(id_pr+2) % 5] != comiendo))
        csignal(requiereTenedor [id_pr+1]);
    /* le da una oportunidad de comer al anterior vecino (número menor) */
    else if ((estado[(id_pr-1) % 5] == hambriento)
     && (estado[(id_pr-2) % 5] != comiendo))
        csignal(requiereTenedor [id_pr-1]);
}
```

```
void filosofo[k=0 hasta 4]                                  /* los cinco clientes filósofos */
{
    while (verdadero)
    {
        <piensa>;
        obtiene_tenedores(k);              /* cliente solicita dos tenedores vía el monitor */
        <come espaguetis>;
        libera_tenedores();                   /* cliente libera tenedores vía el monitor */
    }
}
```

Figura 6.17. Otra solución al problema de los filósofos comensales usando un monitor.

Hilo 1	Hilo 2
`a = 3 ;`	—
`mb () ;`	—
`b = 4 ;`	`c = b ;`
—	`rmb () ;`
—	`d = a ;`

¿Qué errores puede evitar la utilización de las barreras de memoria?

PARTE III

MEMORIA

Uno de los aspectos más complejos del diseño de los sistemas operativos es la gestión de la memoria. Aunque el coste de la memoria ha caído dramáticamente, y como resultado, el tamaño de la memoria principal en las máquinas modernas ha crecido, alcanzando rangos en torno a los gigabytes, nunca hay suficientemente memoria principal para contener todos los programas y estructuras de datos necesarias para los procesos activos y el sistema operativo. Análogamente, una tarea central del sistema operativo es gestionar la memoria, lo que implica traer y llevar bloques de datos de memoria secundaria. Sin embargo, la E/S es una operación lenta, y la velocidad relativa al tiempo de ciclo de una instrucción del procesador aumenta más y más con cada año que pasa. Para mantener ocupado al procesador o procesadores y, por tanto, mantener la eficiencia, el sistema operativo debe gestionar de forma inteligente la transferencia entre memoria principal y secundaria para minimizar el efecto de la E/S en el rendimiento.

ÍNDICE PARA LA PARTE TRES

CAPÍTULO 7. GESTIÓN DE LA MEMORIA

El Capítulo 7 proporciona una descripción básica de los mecanismos fundamentales utilizados en la gestión de la memoria. Primero, se resumen los requisitos básicos de cualquier esquema de gestión de la memoria. A continuación se introduce el uso de particionamiento de la memoria. Esta técnica no es muy utilizada, excepto en casos especiales, tales como la gestión de la memoria del núcleo. Sin embargo, una revisión del particionamiento de la memoria ilumina muchos de los aspectos de diseño relacionados con la gestión de la memoria. El resto del capítulo trata sobre dos técnicas que forman los elementos básicos de prácticamente todos los sistemas de gestión de la memoria: la paginación y la segmentación.

CAPÍTULO 8. MEMORIA VIRTUAL

La memoria virtual, basada en el uso de paginación o la combinación de paginación y segmentación, es una técnica casi universal para la gestión de la memoria en las máquinas contemporáneas. La memoria virtual es un esquema transparente a los procesos, que permite que cada proceso se comporte

como si tuviera memoria ilimitada a su disposición. Para lograr esto, el sistema operativo crea por cada proceso un espacio de direcciones virtual, o memoria virtual, en disco. Parte de la memoria virtual se trae a memoria principal real cuando se necesita. De esta forma, muchos procesos pueden compartir una cantidad relativamente pequeña de memoria principal. Para que la memoria virtual trabaje de forma efectiva, se necesita que los mecanismos de hardware lleven a cabo funciones de paginación y segmentación básicas, tales como traducción de direcciones virtuales a reales. El Capítulo 8 comienza con una descripción de estos mecanismos hardware. El resto del capítulo se dedica a aspectos de diseño del sistema operativo relacionados con la memoria virtual.

C A P Í T U L O 7

Gestión de la memoria

◆ ◆

En un sistema monoprogramado, la memoria se divide en dos partes: una parte para el sistema operativo (monitor residente, núcleo) y una parte para el programa actualmente en ejecución. En un sistema multiprogramado, la parte de «usuario» de la memoria se debe subdividir posteriormente para acomodar múltiples procesos. El sistema operativo es el encargado de la tarea de subdivisión y a esta tarea se le denomina ***gestión de la memoria****.*

Una gestión de la memoria efectiva es vital en un sistema multiprogramado. Si sólo unos pocos procesos se encuentran en memoria, entonces durante una gran parte del tiempo todos los procesos esperarían por operaciones de E/S y el procesador estaría ocioso. Por tanto, es necesario asignar la memoria para asegurar una cantidad de procesos listos que consuman el tiempo de procesador disponible.

Comenzaremos este capítulo con una descripción de los requisitos que la gestión de la memoria pretende satisfacer. A continuación, se mostrará la tecnología de la gestión de la memoria, analizando una variedad de esquemas simples que se han utilizado. El capítulo se enfoca en el requisito de que un programa se debe cargar en memoria principal para ejecutarse. Esta discusión introduce algunos de los principios fundamentales de la gestión de la memoria.

◆ ◆

7.1. REQUISITOS DE LA GESTIÓN DE LA MEMORIA

Mientras se analizan varios mecanismos y políticas asociados con la gestión de la memoria, es útil mantener en mente los requisitos que la gestión de la memoria debe satisfacer. [LIST93] sugiere cinco requisitos:

- Reubicación.
- Protección.
- Compartición.
- Organización lógica.
- Organización física.

REUBICACIÓN

En un sistema multiprogramado, la memoria principal disponible se comparte generalmente entre varios procesos. Normalmente, no es posible que el programador sepa anticipadamente qué programas residirán en memoria principal en tiempo de ejecución de su programa. Adicionalmente, sería bueno poder intercambiar procesos en la memoria principal para maximizar la utilización del procesador, proporcionando un gran número de procesos para la ejecución. Una vez que un programa se ha llevado al disco, sería bastante limitante tener que colocarlo en la misma región de memoria principal donde se hallaba anteriormente, cuando éste se trae de nuevo a la memoria. Por el contrario, podría ser necesario **reubicar** el proceso a un área de memoria diferente.

Por tanto, no se puede conocer de forma anticipada dónde se va a colocar un programa y se debe permitir que los programas se puedan mover en la memoria principal, debido al intercambio o *swap*. Estos hechos ponen de manifiesto algunos aspectos técnicos relacionados con el direccionamiento, como se muestra en la Figura 7.1. La figura representa la imagen de un proceso. Por razones de simplicidad, se asumirá que la imagen de un proceso ocupa una región contigua de la memoria principal. Claramente, el sistema operativo necesitará conocer la ubicación de la información de control

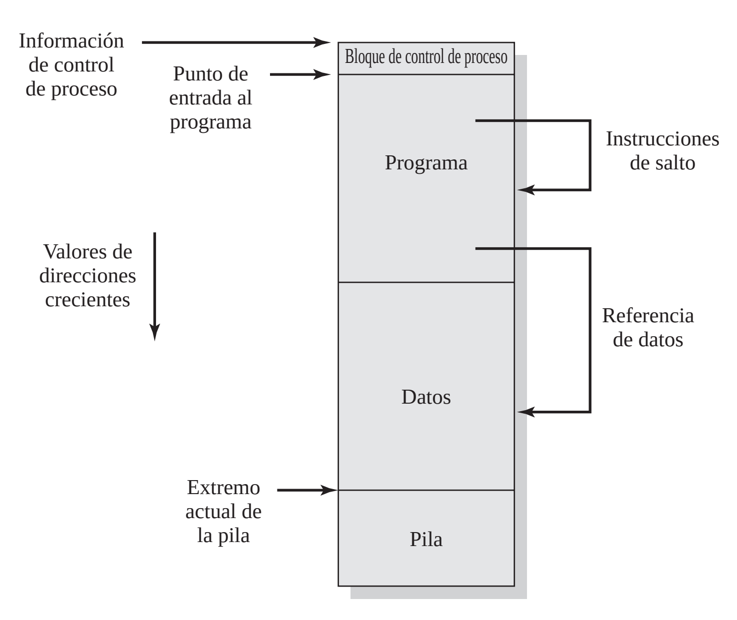


Figura 7.1. Requisitos de direccionamiento para un proceso.

del proceso y de la pila de ejecución, así como el punto de entrada que utilizará el proceso para iniciar la ejecución. Debido a que el sistema operativo se encarga de gestionar la memoria y es responsable de traer el proceso a la memoria principal, estas direcciones son fáciles de adquirir. Adicionalmente, sin embargo, el procesador debe tratar con referencias de memoria dentro del propio programa. Las instrucciones de salto contienen una dirección para referenciar la instrucción que se va a ejecutar a continuación. Las instrucciones de referencia de los datos contienen la dirección del byte o palabra de datos referenciados. De alguna forma, el hardware del procesador y el software del sistema operativo deben poder traducir las referencias de memoria encontradas en el código del programa en direcciones de memoria físicas, que reflejan la ubicación actual del programa en la memoria principal.

PROTECCIÓN

Cada proceso debe protegerse contra interferencias no deseadas por parte de otros procesos, sean accidentales o intencionadas. Por tanto, los programas de otros procesos no deben ser capaces de referenciar sin permiso posiciones de memoria de un proceso, tanto en modo lectura como escritura. Por un lado, lograr los requisitos de la reubicación incrementa la dificultad de satisfacer los requisitos de protección. Más aún, la mayoría de los lenguajes de programación permite el cálculo dinámico de direcciones en tiempo de ejecución (por ejemplo, calculando un índice de posición en un vector o un puntero a una estructura de datos). Por tanto, todas las referencias de memoria generadas por un proceso deben comprobarse en tiempo de ejecución para poder asegurar que se refieren sólo al espacio de memoria asignado a dicho proceso. Afortunadamente, se verá que los mecanismos que dan soporte a la reasignación también dan soporte al requisito de protección.

Normalmente, un proceso de usuario no puede acceder a cualquier porción del sistema operativo, ni al código ni a los datos. De nuevo, un programa de un proceso no puede saltar a una instrucción de otro proceso. Sin un trato especial, un programa de un proceso no puede acceder al área de datos de otro proceso. El procesador debe ser capaz de abortar tales instrucciones en el punto de ejecución.

Obsérvese que los requisitos de protección de memoria deben ser satisfechos por el procesador (hardware) en lugar del sistema operativo (software). Esto es debido a que el sistema operativo no puede anticipar todas las referencias de memoria que un programa hará. Incluso si tal anticipación fuera posible, llevaría demasiado tiempo calcularlo para cada programa a fin de comprobar la violación de referencias de la memoria. Por tanto, sólo es posible evaluar la permisibilidad de una referencia (acceso a datos o salto) en tiempo de ejecución de la instrucción que realiza dicha referencia. Para llevar a cabo esto, el hardware del procesador debe tener esta capacidad.

COMPARTICIÓN

Cualquier mecanismo de protección debe tener la flexibilidad de permitir a varios procesos acceder a la misma porción de memoria principal. Por ejemplo, si varios programas están ejecutando el mismo programa, es ventajoso permitir que cada proceso pueda acceder a la misma copia del programa en lugar de tener su propia copia separada. Procesos que estén cooperando en la misma tarea podrían necesitar compartir el acceso a la misma estructura de datos. Por tanto, el sistema de gestión de la memoria debe permitir el acceso controlado a áreas de memoria compartidas sin comprometer la protección esencial. De nuevo, se verá que los mecanismos utilizados para dar soporte a la reubicación soportan también capacidades para la compartición.

ORGANIZACIÓN LÓGICA

Casi invariablemente, la memoria principal de un computador se organiza como un espacio de almacenamiento lineal o unidimensional, compuesto por una secuencia de bytes o palabras. A nivel físico, la memoria secundaria está organizada de forma similar. Mientras que esta organización es similar al hardware real de la máquina, no se corresponde a la forma en la cual los programas se construyen normalmente. La mayoría de los programas se organizan en módulos, algunos de los cuales no se pueden modificar (sólo lectura, sólo ejecución) y algunos de los cuales contienen datos que se pueden modificar. Si el sistema operativo y el hardware del computador pueden tratar de forma efectiva los programas de usuarios y los datos en la forma de módulos de algún tipo, entonces se pueden lograr varias ventajas:

1. Los módulos se pueden escribir y compilar independientemente, con todas las referencias de un módulo desde otro resueltas por el sistema en tiempo de ejecución.
2. Con una sobrecarga adicional modesta, se puede proporcionar diferentes grados de protección a los módulos (sólo lectura, sólo ejecución).
3. Es posible introducir mecanismos por los cuales los módulos se pueden compartir entre los procesos. La ventaja de proporcionar compartición a nivel de módulo es que se corresponde con la forma en la que el usuario ve el problema, y por tanto es fácil para éste especificar la compartición deseada.

La herramienta que más adecuadamente satisface estos requisitos es la segmentación, que es una de las técnicas de gestión de la memoria exploradas en este capítulo.

ORGANIZACIÓN FÍSICA

Como se discute en la Sección 1.5, la memoria del computador se organiza en al menos dos niveles, conocidos como memoria principal y memoria secundaria. La memoria principal proporciona acceso

rápido a un coste relativamente alto. Adicionalmente, la memoria principal es volátil; es decir, no proporciona almacenamiento permanente. La memoria secundaria es más lenta y más barata que la memoria principal y normalmente no es volátil. Por tanto, la memoria secundaria de larga capacidad puede proporcionar almacenamiento para programas y datos a largo plazo, mientras que una memoria principal más pequeña contiene programas y datos actualmente en uso.

En este esquema de dos niveles, la organización del flujo de información entre la memoria principal y secundaria supone una de las preocupaciones principales del sistema. La responsabilidad para este flujo podría asignarse a cada programador en particular, pero no es practicable o deseable por dos motivos:

1. La memoria principal disponible para un programa más sus datos podría ser insuficiente. En este caso, el programador debería utilizar una técnica conocida como **superposición** (***overlaying***), en la cual los programas y los datos se organizan de tal forma que se puede asignar la misma región de memoria a varios módulos, con un programa principal responsable para intercambiar los módulos entre disco y memoria según las necesidades. Incluso con la ayuda de herramientas de compilación, la programación con *overlays* malgasta tiempo del programador.
2. En un entorno multiprogramado, el programador no conoce en tiempo de codificación cuánto espacio estará disponible o dónde se localizará dicho espacio.

Por tanto, está claro que la tarea de mover la información entre los dos niveles de la memoria debería ser una responsabilidad del sistema. Esta tarea es la esencia de la gestión de la memoria.

7.2. PARTICIONAMIENTO DE LA MEMORIA

La operación principal de la gestión de la memoria es traer los procesos a la memoria principal para que el procesador los pueda ejecutar. En casi todos los sistemas multiprogramados modernos, esto implica el uso de un esquema sofisticado denominado memoria virtual. Por su parte, la memoria virtual se basa en una o ambas de las siguientes técnicas básicas: segmentación y paginación. Antes de fijarse en estas técnicas de memoria virtual, se debe preparar el camino, analizando técnicas más sencillas que no utilizan memoria virtual (Tabla 7.1). Una de estas técnicas, el particionamiento, se ha utilizado en algunas variantes de ciertos sistemas operativos ahora obsoletos. Las otras dos técnicas, paginación sencilla y segmentación sencilla, no son utilizadas de forma aislada. Sin embargo, quedará más clara la discusión de la memoria virtual si se analizan primero estas dos técnicas sin tener en cuenta consideraciones de memoria virtual.

PARTICIONAMIENTO FIJO

En la mayoría de los esquemas para gestión de la memoria, se puede asumir que el sistema operativo ocupa alguna porción fija de la memoria principal y que el resto de la memoria principal está disponible para múltiples procesos. El esquema más simple para gestionar la memoria disponible es repartirla en regiones con límites fijos.

Tamaños de partición

La Figura 7.2 muestra ejemplos de dos alternativas para el particionamiento fijo. Una posibilidad consiste en hacer uso de particiones del mismo tamaño. En este caso, cualquier proceso cuyo tamaño

Tabla 7.1. Técnicas de gestión de memoria.

Técnica	Descripción	Virtudes	Defectos
Particionamiento fijo	La memoria principal se divide en particiones estáticas en tiempo de generación del sistema. Un proceso se puede cargar en una partición con igual o superior tamaño.	Sencilla de implementar; poca sobrecarga para el sistema operativo.	Uso ineficiente de la memoria, debido a la fragmentación interna; debe fijarse el número máximo de procesos activos.
Particionamiento dinámico	Las particiones se crean de forma dinámica, de tal forma que cada proceso se carga en una partición del mismo tamaño que el proceso.	No existe fragmentación interna; uso más eficiente de memoria principal.	Uso ineficiente del procesador, debido a la necesidad de compactación para evitar la fragmentación externa.
Paginación sencilla	La memoria principal se divide en marcos del mismo tamaño. Cada proceso se divide en páginas del mismo tamaño que los marcos. Un proceso se carga a través de la carga de todas sus páginas en marcos disponibles, no necesariamente contiguos.	No existe fragmentación externa.	Una pequeña cantidad de fragmentación interna.
Segmentación sencilla	Cada proceso se divide en segmentos. Un proceso se carga cargando todos sus segmentos en particiones dinámicas, no necesariamente contiguas.	No existe fragmentación interna; mejora la utilización de la memoria y reduce la sobrecargada respecto al particionamiento dinámico.	Fragmentación externa.
Paginación con memoria virtual	Exactamente igual que la paginación sencilla, excepto que no es necesario cargar todas las páginas de un proceso. Las páginas no residentes se traen bajo demanda de forma automática.	No existe fragmentación externa; mayor grado de multiprogramación; gran espacio de direcciones virtuales.	Sobrecarga por la gestión compleja de la memoria.
Segmentación con memoria virtual	Exactamente igual que la segmentación, excepto que no es necesario cargar todos los segmentos de un proceso. Los segmentos no residentes se traen bajo demanda de forma automática.	No existe fragmentación interna; mayor grado de multiprogramación; gran espacio de direcciones virtuales; soporte a protección y compartición.	Sobrecarga por la gestión compleja de la memoria.

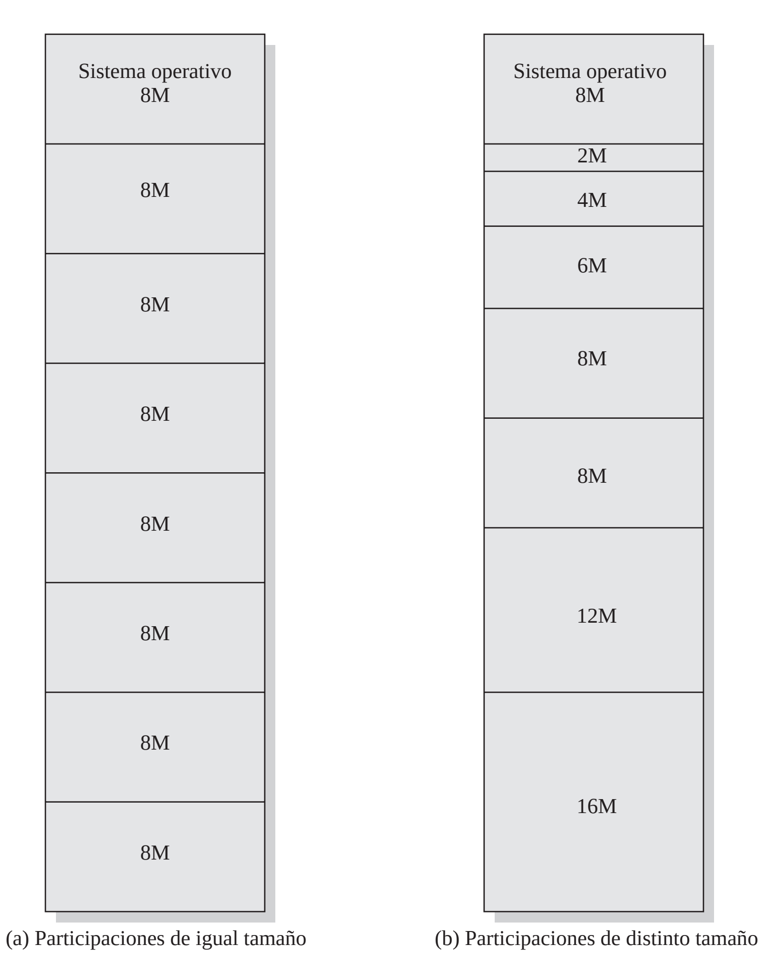


Figura 7.2. Ejemplo de particionamiento fijo de una memoria de 64 Mbytes.

es menor o igual que el tamaño de partición puede cargarse en cualquier partición disponible. Si todas las particiones están llenas y no hay ningún proceso en estado Listo o Ejecutando, el sistema operativo puede mandar a *swap* a un proceso de cualquiera de las particiones y cargar otro proceso, de forma que el procesador tenga trabajo que realizar.

Existen dos dificultades con el uso de las particiones fijas del mismo tamaño:

- Un programa podría ser demasiado grande para caber en una partición. En este caso, el programador debe diseñar el programa con el uso de *overlays*, de forma que sólo se necesite una porción del programa en memoria principal en un momento determinado. Cuando se necesita un módulo que no está presente, el programa de usuario debe cargar dicho módulo en la partición del programa, superponiéndolo (*overlaying*) a cualquier programa o datos que haya allí.
- La utilización de la memoria principal es extremadamente ineficiente. Cualquier programa, sin importar lo pequeño que sea, ocupa una partición entera. En el ejemplo, podría haber un programa cuya longitud es menor que 2 Mbytes; ocuparía una partición de 8 Mbytes cuando se lleva a la memoria. Este fenómeno, en el cual hay espacio interno malgastado debido al hecho

de que el bloque de datos cargado es menor que la partición, se conoce con el nombre de **fragmentación interna**.

Ambos problemas se pueden mejorar, aunque no resolver, utilizando particiones de tamaño diferente (Figura 7.2b). En este ejemplo, los programas de 16 Mbytes se pueden acomodar sin *overlays*. Las particiones más pequeñas de 8 Mbytes permiten que los programas más pequeños se puedan acomodar sin menor fragmentación interna.

Algoritmo de ubicación

Con particiones del mismo tamaño, la ubicación de los procesos en memoria es trivial. En cuanto haya una partición disponible, un proceso se carga en dicha partición. Debido a que todas las particiones son del mismo tamaño, no importa qué partición se utiliza. Si todas las particiones se encuentran ocupadas por procesos que no están listos para ejecutar, entonces uno de dichos procesos debe llevarse a disco para dejar espacio para un nuevo proceso. Cuál de los procesos se lleva a disco es una decisión de planificación; este tema se describe en la Parte Cuatro.

Con particiones de diferente tamaño, hay dos formas posibles de asignar los procesos a las particiones. La forma más sencilla consiste en asignar cada proceso a la partición más pequeña dentro de la cual cabe[1]. En este caso, se necesita una cola de planificación para cada partición, que mantenga procesos en disco destinados a dicha partición (Figura 7.3a). La ventaja de esta técnica es que los procesos siempre se asignan de tal forma que se minimiza la memoria malgastada dentro de una partición (fragmentación interna).

Aunque esta técnica parece óptima desde el punto de vista de una partición individual, no es óptima desde el punto de vista del sistema completo. En la Figura 7.2b, por ejemplo, se considera un caso en el que no haya procesos con un tamaño entre 12 y 16M en un determinado instante de tiempo. En este caso, la partición de 16M quedará sin utilizarse, incluso aunque se puede asignar dicha partición a algunos procesos más pequeños. Por tanto, una técnica óptima sería emplear una única cola para todos los procesos (Figura 7.3b). En el momento de cargar un proceso en la memoria principal, se selecciona la partición más pequeña disponible que puede albergar dicho proceso. Si todas las particiones están ocupadas, se debe llevar a cabo una decisión para enviar a *swap* a algún proceso. Tiene preferencia a la hora de ser expulsado a disco el proceso que ocupe la partición más pequeña que pueda albergar al proceso entrante. Es también posible considerar otros factores, como la prioridad o una preferencia por expulsar a disco procesos bloqueados frente a procesos listos.

El uso de particiones de distinto tamaño proporciona un grado de flexibilidad frente a las particiones fijas. Adicionalmente, se puede decir que los esquemas de particiones fijas son relativamente sencillos y requieren un soporte mínimo por parte del sistema operativo y una sobrecarga de procesamiento mínimo. Sin embargo, tiene una serie de desventajas:

- El número de particiones especificadas en tiempo de generación del sistema limita el número de proceso activos (no suspendidos) del sistema.
- Debido a que los tamaños de las particiones son preestablecidos en tiempo de generación del sistema, los trabajos pequeños no utilizan el espacio de las particiones eficientemente. En un entorno donde el requisito de almacenamiento principal de todos los trabajos se conoce de an-

---

[1] Se asume que se conoce el tamaño máximo de memoria que un proceso requerirá. No siempre es el caso. Si no se sabe lo que un proceso puede ocupar, la única alternativa es un esquema de *overlays* o el uso de memoria virtual.

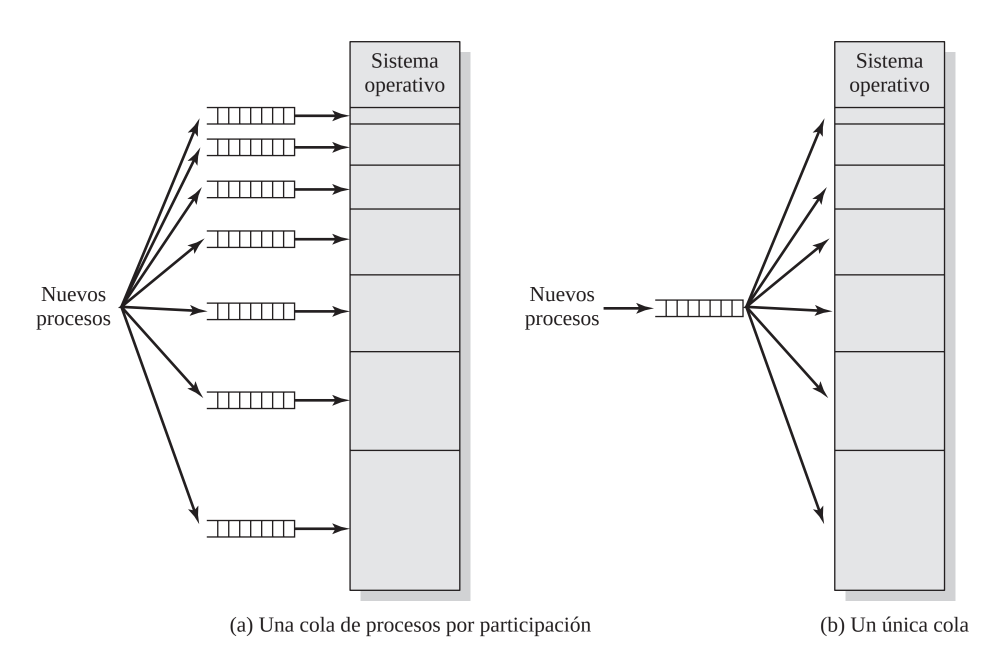


Figura 7.3. Asignación de memoria para particionamiento fijo.

temano, esta técnica puede ser razonable, pero en la mayoría de los casos, se trata de una técnica ineficiente.

El uso de particionamiento fijo es casi desconocido hoy en día. Un ejemplo de un sistema operativo exitoso que sí utilizó esta técnica fue un sistema operativo de los primeros *mainframes* de IBM, el sistema operativo OS/MFT (*Multiprogramming with a Fixed Number of Tasks*; Multiprogramado con un número fijo de tareas).

PARTICIONAMIENTO DINÁMICO

Para vencer algunas de las dificultades con particionamiento fijo, se desarrolló una técnica conocida como particionamiento dinámico. De nuevo, esta técnica se ha sustituido por técnicas de gestión de memoria más sofisticadas. Un sistema operativo importante que utilizó esta técnica fue el sistema operativo de *mainframes* de IBM, el sistema operativo OS/MVT (*Multiprogramming with a Variable Number of Tasks*; Multiprogramado con un número variable de tareas).

Con particionamiento dinámico, las particiones son de longitud y número variable. Cuando se lleva un proceso a la memoria principal, se le asigna exactamente tanta memoria como requiera y no más. Un ejemplo, que utiliza 64 Mbytes de memoria principal, se muestra en la Figura 7.4. Inicialmente, la memoria principal está vacía, excepto por el sistema operativo (a). Los primeros tres procesos se cargan justo donde el sistema operativo finaliza y ocupando el espacio justo para cada proceso (b, c, d). Esto deja un «hueco» al final de la memoria que es demasiado pequeño para un cuarto proceso. En algún momento, ninguno de los procesos que se encuentran en memoria está disponible. El sistema operativo lleva el proceso 2 al disco (e), que deja suficiente espacio para cargar un nuevo proceso, el proceso 4 (f). Debido a que el proceso 4 es más pequeño que el proceso 2, se crea otro pequeño hueco. Posteriormente, se alcanza un punto en el cual ninguno de los procesos de la memoria principal está listo, pero el proceso 2, en estado Listo-Suspendido, está disponible. Porque no hay es-

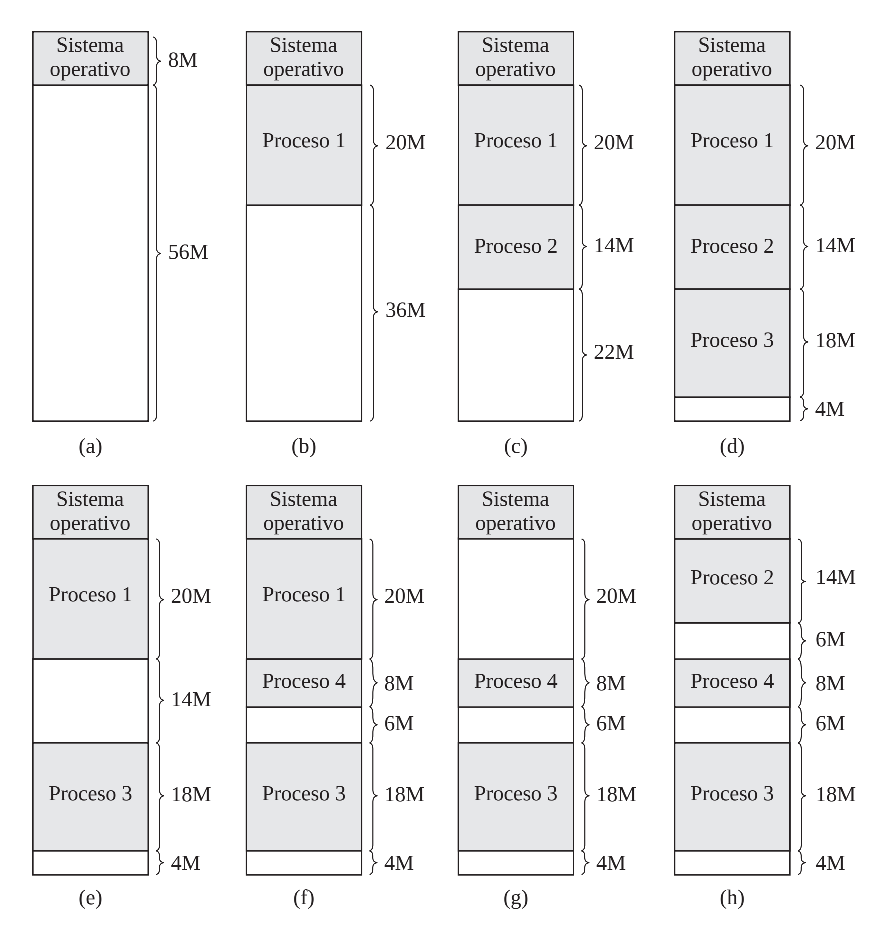


Figura 7.4. El efecto del particionamiento dinámico.

pacio suficiente en la memoria para el proceso 2, el sistema operativo lleva a disco el proceso 1 (g) y lleva a la memoria el proceso 2 (h).

Como muestra este ejemplo, el método comienza correctamente, pero finalmente lleva a una situación en la cual existen muchos huecos pequeños en la memoria. A medida que pasa el tiempo, la memoria se fragmenta cada vez más y la utilización de la memoria se decrementa. Este fenómeno se conoce como **fragmentación externa**, indicando que la memoria que es externa a todas las particiones se fragmenta de forma incremental, por contraposición a lo que ocurre con la fragmentación interna, descrita anteriormente.

Una técnica para eliminar la fragmentación externa es la **compactación**: de vez en cuando, el sistema operativo desplaza los procesos en memoria, de forma que se encuentren contiguos y de este modo toda la memoria libre se encontrará unida en un bloque. Por ejemplo, en la Figura 7.4R, la compactación permite obtener un bloque de memoria libre de longitud 16M. Esto sería suficiente para cargar un proceso adicional. La desventaja de la compactación es el hecho de que se trata de un procedimiento que consume tiempo y malgasta tiempo de procesador. Obsérvese que la compactación requiere la capacidad de reubicación dinámica. Es decir, debe ser posible mover un programa desde una región a otra en la memoria principal sin invalidar las referencias de la memoria de cada programa (*véase* Apéndice 7A).

Algoritmo de ubicación

Debido a que la compactación de memoria consume una gran cantidad de tiempo, el diseñador del sistema operativo debe ser inteligente a la hora de decidir cómo asignar la memoria a los procesos (cómo eliminar los huecos). A la hora de cargar o intercambiar un proceso a la memoria principal, y siempre que haya más de un bloque de memoria libre de suficiente tamaño, el sistema operativo debe decidir qué bloque libre asignar.

Tres algoritmos de colocación que pueden considerarse son mejor-ajuste (*best-fit*), primer-ajuste(*first-fit*) y siguiente-ajuste (*next-fit*). Todos, por supuesto, están limitados a escoger entre los bloques libres de la memoria principal que son iguales o más grandes que el proceso que va a llevarse a la memoria. **Mejor-ajuste** escoge el bloque más cercano en tamaño a la petición. **Primer-ajuste** comienza a analizar la memoria desde el principio y escoge el primer bloque disponible que sea suficientemente grande. **Siguiente-ajuste** comienza a analizar la memoria desde la última colocación y elige el siguiente bloque disponible que sea suficientemente grande.

La Figura 7.5a muestra un ejemplo de configuración de memoria después de un número de colocaciones e intercambios a disco. El último bloque que se utilizó fue un bloque de 22 Mbytes del cual se crea una partición de 14 Mbytes. La Figura 7.5b muestra la diferencia entre los algoritmos de mejor-, primer- y siguiente- ajuste a la hora de satisfacer una petición de asignación de 16 Mbytes. Me-

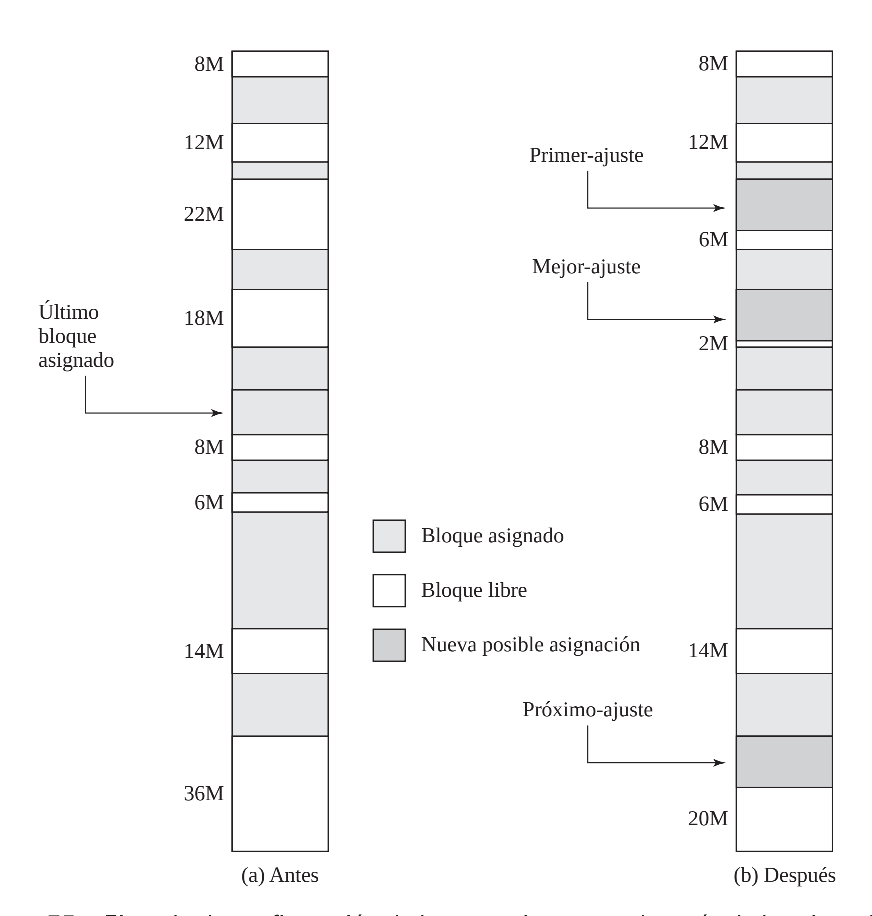


Figura 7.5. Ejemplo de configuración de la memoria antes y después de la asignación de un bloque de 16 Mbytes.

jor-ajuste busca la lista completa de bloques disponibles y hace uso del bloque de 18 Mbytes, dejando un fragmento de 2 Mbytes. *Primer-ajuste* provoca un fragmento de 6 Mbytes, y *siguiente-ajuste* provoca un fragmento de 20 Mbytes.

Cuál de estas técnicas es mejor depende de la secuencia exacta de intercambio de procesos y del tamaño de dichos procesos. Sin embargo, se pueden hacer algunos comentarios (*véase* también [BREN89], [SHOR75] y [BAYS77]). El algoritmo primer-ajuste no es sólo el más sencillo, sino que normalmente es también el mejor y más rápido. El algoritmo siguiente-ajuste tiende a producir resultados ligeramente peores que el primer-ajuste. El algoritmo siguiente-ajuste llevará más frecuentemente a una asignación de un bloque libre al final de la memoria. El resultado es que el bloque más grande de memoria libre, que normalmente aparece al final del espacio de la memoria, se divide rápidamente en pequeños fragmentos. Por tanto, en el caso del algoritmo siguiente-ajuste se puede requerir más frecuentemente la compactación. Por otro lado, el algoritmo primer-ajuste puede dejar el final del espacio de almacenamiento con pequeñas particiones libres que necesitan buscarse en cada paso del primer-ajuste siguiente. El algoritmo mejor-ajuste, a pesar de su nombre, su comportamiento normalmente es el peor. Debido a que el algoritmo busca el bloque más pequeño que satisfaga la petición, garantiza que el fragmento que quede sea lo más pequeño posible. Aunque cada petición de memoria siempre malgasta la cantidad más pequeña de la memoria, el resultado es que la memoria principal se queda rápidamente con bloques demasiado pequeños para satisfacer las peticiones de asignación de la memoria. Por tanto, la compactación de la memoria se debe realizar más frecuentemente que con el resto de los algoritmos.

Algoritmo de reemplazamiento

En un sistema multiprogramado que utiliza particionamiento dinámico, puede haber un momento en el que todos los procesos de la memoria principal estén en estado bloqueado y no haya suficiente memoria para un proceso adicional, incluso después de llevar a cabo una compactación. Para evitar malgastar tiempo de procesador esperando a que un proceso se desbloquee, el sistema operativo intercambiará alguno de los procesos entre la memoria principal y disco para hacer sitio a un nuevo proceso o para un proceso que se encuentre en estado Listo-Suspendido. Por tanto, el sistema operativo debe escoger qué proceso reemplazar. Debido a que el tema de los algoritmos de reemplazo se contemplará en detalle respecto a varios esquemas de la memoria virtual, se pospone esta discusión hasta entonces.

SISTEMA BUDDY

Ambos esquemas de particionamiento, fijo y dinámico, tienen desventajas. Un esquema de particionamiento fijo limita el número de procesos activos y puede utilizar el espacio ineficientemente si existe un mal ajuste entre los tamaños de partición disponibles y los tamaños de los procesos. Un esquema de particionamiento dinámico es más complejo de mantener e incluye la sobrecarga de la compactación. Un compromiso interesante es el sistema *buddy* ([KNUT97], [PETE77]).

En un sistema *buddy*, los bloques de memoria disponibles son de tamaño 2^k, $L \leq K \leq U$, donde

2^L = bloque de tamaño más pequeño asignado

2^U = bloque de tamaño mayor asignado; normalmente 2^U es el tamaño de la memoria completa disponible

Para comenzar, el espacio completo disponible se trata como un único bloque de tamaño 2^U. Si se realiza una petición de tamaño *s*, tal que $2^{U-1} < s \leq 2^U$, se asigna el bloque entero. En otro caso, el blo-

que se divide en dos bloques *buddy* iguales de tamaño 2^{U-1}. Si $2^{U-2} < s \leq 2^{U-1}$, entonces se asigna la petición a uno de los otros dos bloques. En otro caso, uno de ellos se divide por la mitad de nuevo. Este proceso continúa hasta que el bloque más pequeño mayor o igual que *s* se genera y se asigna a la petición. En cualquier momento, el sistema *buddy* mantiene una lista de huecos (bloques sin asignar) de cada tamaño 2^i. Un hueco puede eliminarse de la lista (i+1) dividiéndolo por la mitad para crear dos bloques de tamaño 2^i en la lista *i*. Siempre que un par de bloques de la lista *i* no se encuentren asignados, son eliminados de dicha lista y unidos en un único bloque de la lista (i+1). Si se lleva a cabo una petición de asignación de tamaño *k* tal que $2^{i-1} < k \leq 2^i$, se utiliza el siguiente algoritmo recursivo (de [LIST93]) para encontrar un hueco de tamaño 2^i:

```
void obtener_hueco (int i)
{
    if (i==(U+1))
        <fallo>;
    if (<lista_i vacía>)
    {
        obtener_hueco(i+1);
        <dividir hueco en dos buddies>;
        <colocar buddies en lista_i>;
    }
    <tomar primer hueco de la lista_i>;
}
```

La Figura 7.6 muestra un ejemplo que utiliza un bloque inicial de 1 Mbyte. La primera petición, A, es de 100 Kbytes, de tal forma que se necesita un bloque de 128K. El bloque inicial se divide en dos de 512K. El primero de éstos se divide en dos bloques de 256K, y el primero de éstos se divide en dos de 128K, uno de los cuales se asigna a A. La siguiente petición, B, solicita un bloque de 256K. Dicho bloque está disponible y es asignado. El proceso continúa, provocando divisiones y fusiones de bloques cuando se requiere. Obsérvese que cuando se libera E, se unen dos bloques de 128K en un bloque de 256K, que es inmediatamente unido con su bloque compañero (*buddy*).

La Figura 7.7 muestra una representación en forma de árbol binario de la asignación de bloques inmediatamente después de la petición «Liberar B». Los nodos hoja representan el particionamiento de la memoria actual. Si dos bloques son nodos hoja, entonces al menos uno de ellos debe estar asignado; en otro caso, se unen en un bloque mayor.

El sistema *buddy* es un compromiso razonable para eliminar las desventajas de ambos esquemas de particionamiento, fijo y variable, pero en los sistemas operativos contemporáneos, la memoria virtual basada en paginación y segmentación es superior. Sin embargo, el sistema *buddy* se ha utilizado en sistemas paralelos como una forma eficiente de asignar y liberar programas paralelos (por ejemplo, *véase* [JOHN92]). Una forma modificada del sistema *buddy* se utiliza en la asignación de memoria del núcleo UNIX (descrito en el Capítulo 8).

REUBICACIÓN

Antes de considerar formas de tratar los problemas del particionamiento, se va a aclarar un aspecto relacionado con la colocación de los procesos en la memoria. Cuando se utiliza el esquema de particionamiento fijo de la Figura 7.3a, se espera que un proceso siempre se asigne a la misma partición.

1 bloque de 1 Mbyte	1M
Solicitar 100K	A = 128K | 128K | 256K | 512K
Solicitar 240K	A = 128K | 128K | B = 256K | 512K
Solicitar 64K	A = 128K | C = 64K | 64K | B = 256K | 512K
Solicitar 256K	A = 128K | C = 64K | 64K | B = 256K | D = 256K | 256K
Liberar B	A = 128K | C = 64K | 64K | 256K | D = 256K | 256K
Liberar A	128K | C = 64K | 64K | 256K | D = 256K | 256K
Solicitar 75K	E = 128K | C = 64K | 64K | 256K | D = 256K | 256K
Liberar C	E = 128K | 128K | 256K | D = 256K | 256K
Liberar E	512K | D = 256K | 256K
Liberar D	1M

Figura 7.6. Ejemplo de sistema *Buddy.*

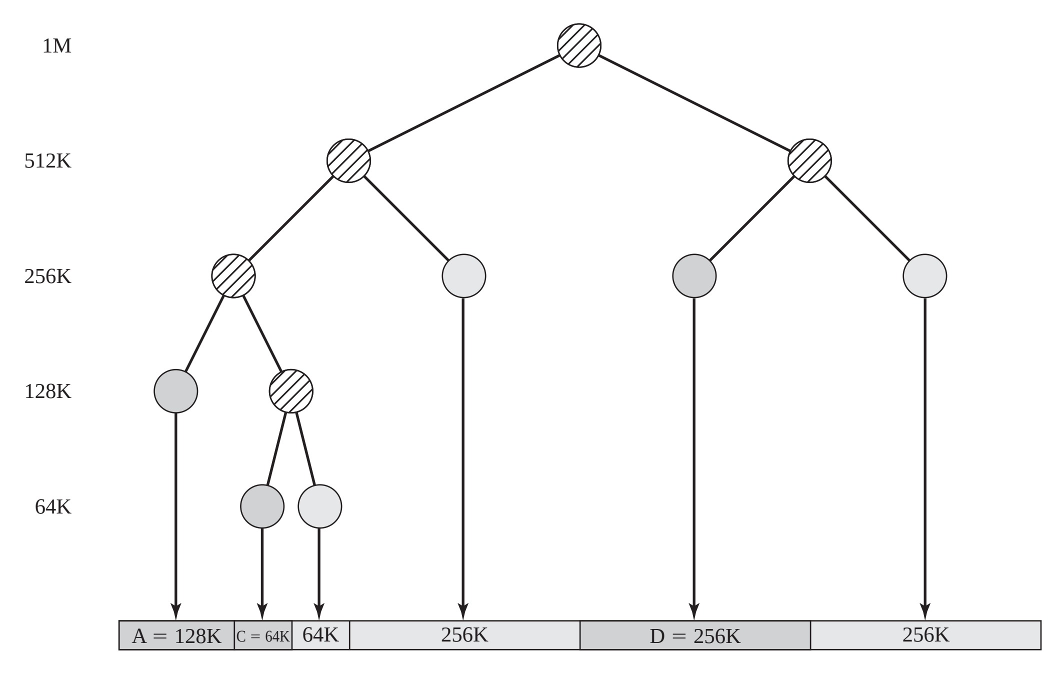


Figura 7.7. Representación en forma de árbol del sistema *Buddy.*

Es decir, sea cual sea la partición seleccionada cuando se carga un nuevo proceso, ésta será la utilizada para el intercambio del proceso entre la memoria y el área de *swap* en disco. En este caso, se utiliza un cargador sencillo, tal y como se describe en el Apéndice 7A: cuando el proceso se carga por

primera vez, todas las referencias de la memoria relativas del código se reemplazan por direcciones de la memoria principal absolutas, determinadas por la dirección base del proceso cargado.

En el caso de particiones de igual tamaño (Figura 7.2), y en el caso de una única cola de procesos para particiones de distinto tamaño (Figura 7.3b), un proceso puede ocupar diferentes particiones durante el transcurso de su ciclo de vida. Cuando la imagen de un proceso se crea por primera vez, se carga en la misma partición de memoria principal. Más adelante, el proceso podría llevarse a disco; cuando se trae a la memoria principal posteriormente, podría asignarse a una partición distinta de la última vez. Lo mismo ocurre en el caso del particionamiento dinámico. Obsérvese en las Figuras 7.4c y h que el proceso 2 ocupa dos regiones diferentes de memoria en las dos ocasiones que se trae a la memoria. Más aún, cuando se utiliza la compactación, los procesos se desplazan mientras están en la memoria principal. Por tanto, las ubicaciones (de las instrucciones y los datos) referenciadas por un proceso no son fijas. Cambiarán cada vez que un proceso se intercambia o se desplaza. Para resolver este problema, se realiza una distinción entre varios tipos de direcciones. Una **dirección lógica** es una referencia a una ubicación de memoria independiente de la asignación actual de datos a la memoria; se debe llevar a cabo una traducción a una dirección física antes de que se alcance el acceso a la memoria. Una **dirección relativa** es un ejemplo particular de dirección lógica, en el que la dirección se expresa como una ubicación relativa a algún punto conocido, normalmente un valor en un registro del procesador. Una **dirección física**, o dirección absoluta, es una ubicación real de la memoria principal.

Los programas que emplean direcciones relativas de memoria se cargan utilizando carga dinámica en tiempo de ejecución (*véase* Apéndice 7A, donde se recoge una discusión). Normalmente, todas las referencias de memoria de los procesos cargados son relativas al origen del programa. Por tanto, se necesita un mecanismo hardware para traducir las direcciones relativas a direcciones físicas de la memoria principal, en tiempo de ejecución de la instrucción que contiene dicha referencia.

La Figura 7.8 muestra la forma en la que se realiza normalmente esta traducción de direcciones. Cuando un proceso se asigna al estado Ejecutando, un registro especial del procesador, algunas veces llamado registro base, carga la dirección inicial del programa en la memoria principal. Existe también un registro «valla», que indica el final de la ubicación del programa; estos valores se establecen cuando el programa se carga en la memoria o cuando la imagen del proceso se lleva a la memoria. A lo largo de la ejecución del proceso, se encuentran direcciones relativas. Éstas incluyen los contenidos del registro de las instrucciones, las direcciones de instrucciones que ocurren en los saltos e instrucciones *call*, y direcciones de datos existentes en instrucciones de carga y almacenamiento. El procesador manipula cada dirección relativa, a través de dos pasos. Primero, el valor del registro base se suma a la dirección relativa para producir una dirección absoluta. Segundo, la dirección resultante se compara con el valor del registro «valla». Si la dirección se encuentra dentro de los límites, entonces se puede llevar a cabo la ejecución de la instrucción. En otro caso, se genera una interrupción, que debe manejar el sistema operativo de algún modo.

El esquema de la Figura 7.8 permite que se traigan a memoria los programas y que se lleven a disco, a lo largo de la ejecución. También proporciona una medida de protección: cada imagen del proceso está aislada mediante los contenidos de los registros base y valla. Además, evita accesos no autorizados por parte de otros procesos.

7.3. PAGINACIÓN

Tanto las particiones de tamaño fijo como variable son ineficientes en el uso de la memoria; los primeros provocan fragmentación interna, los últimos fragmentación externa. Supóngase, sin embargo, que la memoria principal se divide en porciones de tamaño fijo relativamente pequeños, y que cada proceso también se divide en porciones pequeñas del mismo tamaño fijo. A dichas porciones del pro-

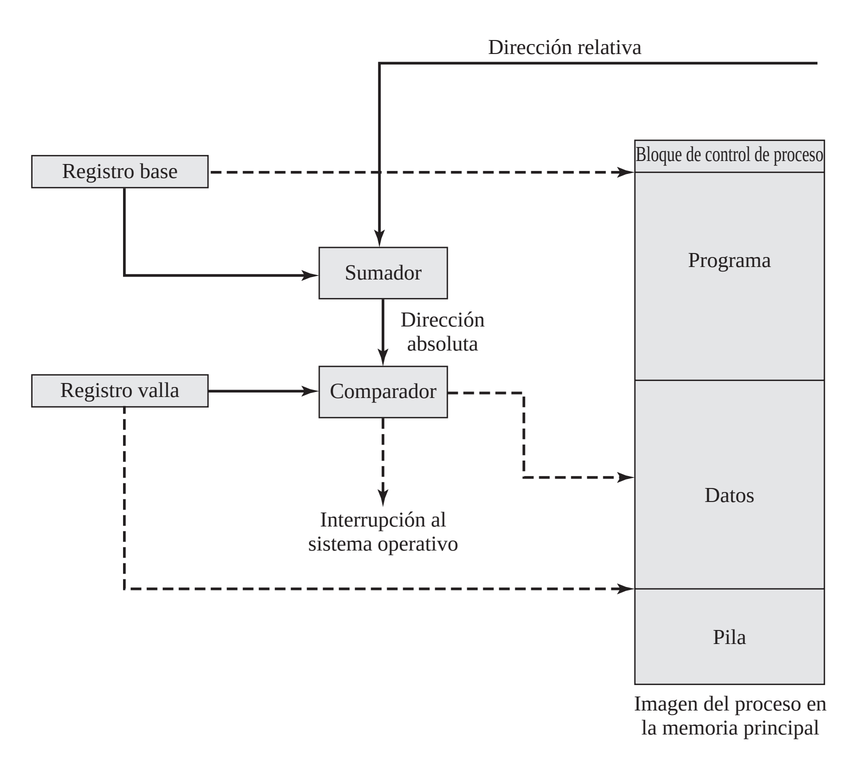


Figura 7.8. Soporte hardware para la reubicación.

ceso, conocidas como **páginas**, se les asigna porciones disponibles de memoria, conocidas como **marcos**, o marcos de páginas. Esta sección muestra que el espacio de memoria malgastado por cada proceso debido a la fragmentación interna corresponde sólo a una fracción de la última página de un proceso. No existe fragmentación externa.

La Figura 7.9 ilustra el uso de páginas y marcos. En un momento dado, algunos de los marcos de la memoria están en uso y algunos están libres. El sistema operativo mantiene una lista de marcos libres. El proceso A, almacenado en disco, está formado por cuatro páginas. En el momento de cargar este proceso, el sistema operativo encuentra cuatro marcos libres y carga las cuatro páginas del proceso A en dichos marcos (Figura 7.9b). El proceso B, formado por tres páginas, y el proceso C, formado por cuatro páginas, se cargan a continuación. En ese momento, el proceso B se suspende y deja la memoria principal. Posteriormente, todos los procesos de la memoria principal se bloquean, y el sistema operativo necesita traer un nuevo proceso, el proceso D, que está formado por cinco páginas.

Ahora supóngase, como en este ejemplo, que no hay suficientes marcos contiguos sin utilizar para ubicar un proceso. ¿Esto evitaría que el sistema operativo cargara el proceso D? La respuesta es no, porque una vez más se puede utilizar el concepto de dirección lógica. Un registro base sencillo de direcciones no basta en esta ocasión. En su lugar, el sistema operativo mantiene una **tabla de páginas** por cada proceso. La tabla de páginas muestra la ubicación del marco por cada página del proceso. Dentro del programa, cada dirección lógica está formada por un número de página y un desplazamiento dentro de la página. Es importante recordar que en el caso de una partición simple, una dirección lógica es la ubicación de una palabra relativa al comienzo del programa; el procesador la traduce en una dirección física. Con paginación, la traducción de direcciones lógicas a físicas las realiza también el hardware del procesador. Ahora el procesador debe conocer cómo acceder a la ta-

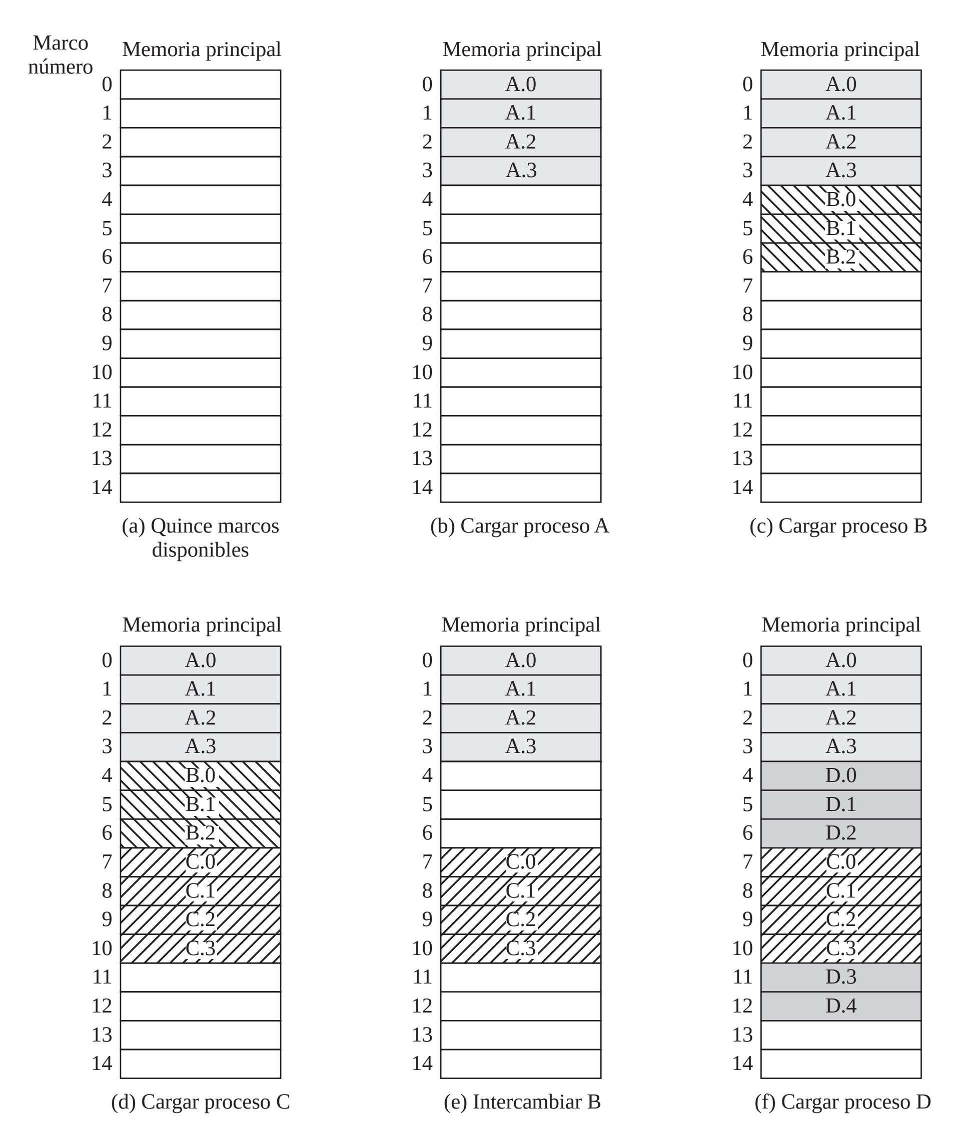


Figura 7.9. Asignación de páginas de proceso a marcos libres.

bla de páginas del proceso actual. Presentado como una dirección lógica (número de página, desplazamiento), el procesador utiliza la tabla de páginas para producir una dirección física (número de marco, desplazamiento).

Continuando con nuestro ejemplo, las cinco páginas del proceso D se cargan en los marcos 4, 5, 6, 11 y 12. La Figura 7.10 muestra las diferentes tablas de páginas en este momento. Una tabla de páginas contiene una entrada por cada página del proceso, de forma que la tabla se indexe fácilmente por el número de página (iniciando en la página 0). Cada entrada de la tabla de páginas contiene el número del marco en la memoria principal, si existe, que contiene la página correspondiente, Adicionalmente, el sistema operativo mantiene una única lista de marcos libres de todos los marcos de la memoria que se encuentran actualmente no ocupados y disponibles para las páginas.

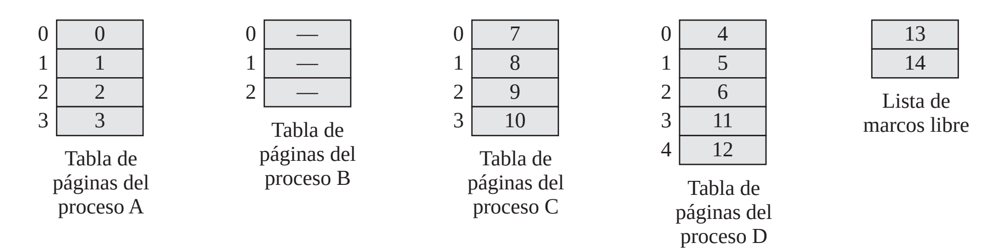


Figura 7.10. Estructuras de datos para el ejemplo de la Figura 7.9 en el instante (f).

Por tanto vemos que la paginación simple, tal y como se describe aquí, es similar al particionamiento fijo. Las diferencias son que, con la paginación, las particiones son bastante pequeñas; un programa podría ocupar más de una partición; y dichas particiones no necesitan ser contiguas.

Para hacer este esquema de paginación conveniente, el tamaño de página y por tanto el tamaño del marco debe ser una potencia de 2. Con el uso de un tamaño de página potencia de 2, es fácil demostrar que la dirección relativa, que se define con referencia al origen del programa, y la dirección lógica, expresada como un número de página y un desplazamiento, es lo mismo. Se muestra un ejemplo en la Figura 7.11. En este ejemplo, se utilizan direcciones de 16 bits, y el tamaño de la página es 1K = 1024 bytes. La dirección relativa 1502, en formato binario, es 0000010111011110. Con una página de tamaño 1K, se necesita un campo de desplazamiento de 10 bits, dejando 6 bits para el número de página. Por tanto, un programa puede estar compuesto por un máximo de 2^6=64 páginas de 1K byte cada una. Como muestra la Figura 7.11, la dirección relativa 1502 corresponde a un desplazamiento de 478 (0111011110) en la página 1 (000001), que forma el mismo número de 16 bits, 0000010111011110.

Las consecuencias de utilizar un tamaño de página que es una potencia de 2 son dobles. Primero, el esquema de direccionamiento lógico es transparente al programador, al ensamblador y al montador. Cada dirección lógica (número de página, desplazamiento) de un programa es idéntica a su dirección relativa. Segundo, es relativamente sencillo implementar una función que ejecute el hardware para llevar a cabo dinámicamente la traducción de direcciones en tiempo de ejecución. Considérese una dirección de $n+m$ bits, donde los n bits de la izquierda corresponden al número de página y los m bits de la derecha corresponden al desplazamiento. En el ejemplo (Figura 7.11b), $n = 6$ y $m = 10$. Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de página como los n bits de la izquierda de la dirección lógica.
- Utilizar el número de página como un índice a tabla de páginas del proceso para encontrar el número de marco, k.
- La dirección física inicial del marco es $k \times 2^m$, y la dirección física del byte referenciado es dicho número más el desplazamiento. Esta dirección física no necesita calcularse; es fácilmente construida concatenando el número de marco al desplazamiento.

En el ejemplo, se parte de la dirección lógica 0000010111011110, que corresponde a la página número 1, desplazamiento 478. Supóngase que esta página reside en el marco de memoria principal 6 = número binario 000110. Por tanto, la dirección física corresponde al marco número 6, desplazamiento 478 = 0001100111011110 (Figura 7.12a).

Resumiendo, con la paginación simple, la memoria principal se divide en muchos marcos pequeños de igual tamaño. Cada proceso se divide en páginas de igual tamaño; los procesos más pequeños requieren menos páginas, los procesos mayores requieren más. Cuando un proceso se trae a la memo-

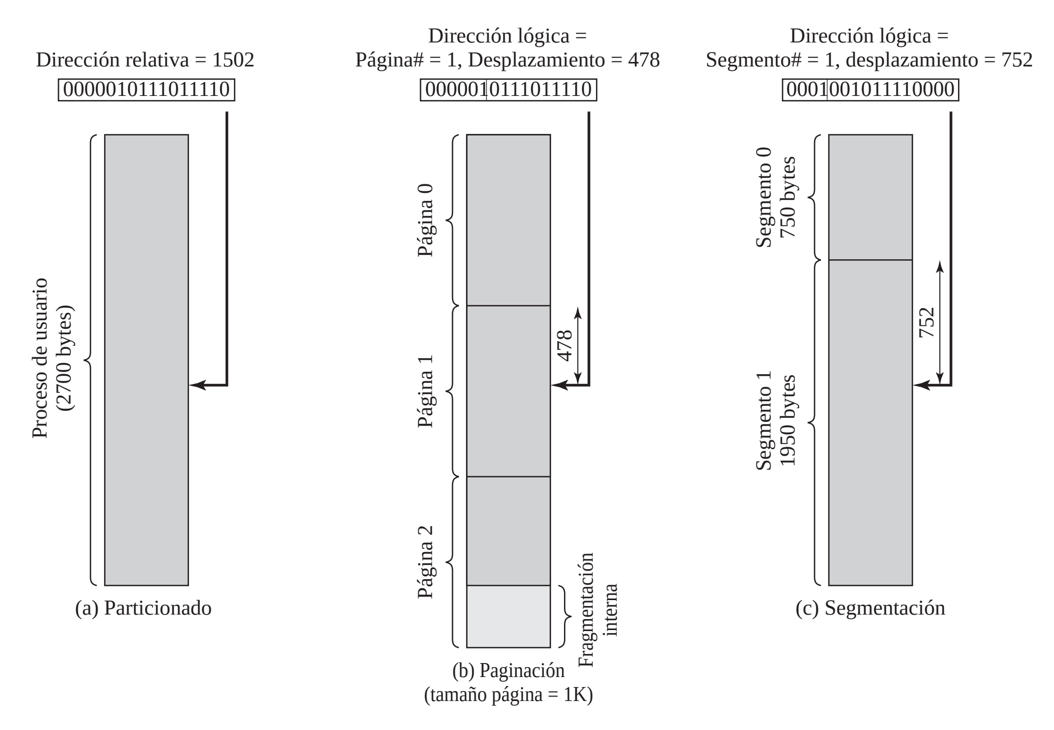


Figura 7.11. Direcciones lógicas.

ria, todas sus páginas se cargan en los marcos disponibles, y se establece una tabla de páginas. Esta técnica resuelve muchos de los problemas inherentes en el particionamiento.

7.4. SEGMENTACIÓN

Un programa de usuario se puede subdividir utilizando segmentación, en la cual el programa y sus datos asociados se dividen en un número de **segmentos**. No se requiere que todos los programas sean de la misma longitud, aunque hay una longitud máxima de segmento. Como en el caso de la paginación, una dirección lógica utilizando segmentación está compuesta por dos partes, en este caso un número de segmento y un desplazamiento.

Debido al uso de segmentos de distinto tamaño, la segmentación es similar al particionamiento dinámico. En la ausencia de un esquema de *overlays* o el uso de la memoria virtual, se necesitaría que todos los segmentos de un programa se cargaran en la memoria para su ejecución. La diferencia, comparada con el particionamiento dinámico, es que con la segmentación un programa podría ocupar más de una partición, y estas particiones no necesitan ser contiguas. La segmentación elimina la fragmentación interna pero, al igual que el particionamiento dinámico, sufre de fragmentación externa. Sin embargo, debido a que el proceso se divide en varias piezas más pequeñas, la fragmentación externa debería ser menor.

Mientras que la paginación es invisible al programador, la segmentación es normalmente visible y se proporciona como una utilidad para organizar programas y datos. Normalmente, el programador o compilador asignará programas y datos a diferentes segmentos. Para los propósitos de la programación modular, los programas o datos se pueden dividir posteriormente en múltiples segmentos. El in-

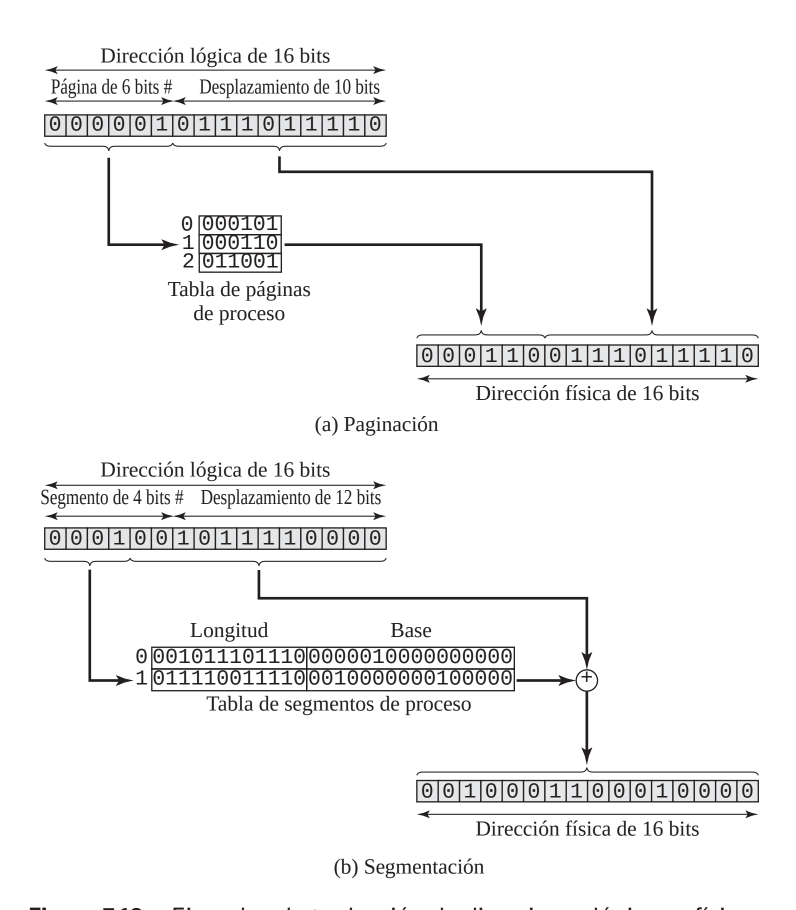


Figura 7.12. Ejemplos de traducción de direcciones lógicas a físicas.

conveniente principal de este servicio es que el programador debe ser consciente de la limitación de tamaño de segmento máximo.

Otra consecuencia de utilizar segmentos de distinto tamaño es que no hay una relación simple entre direcciones lógicas y direcciones físicas. De forma análoga a la paginación, un esquema de segmentación sencillo haría uso de una tabla de segmentos por cada proceso y una lista de bloques libre de memoria principal. Cada entrada de la tabla de segmentos tendría que proporcionar la dirección inicial de la memoria principal del correspondiente segmento. La entrada también debería proporcionar la longitud del segmento, para asegurar que no se utilizan direcciones no válidas. Cuando un proceso entra en el estado Ejecutando, la dirección de su tabla de segmentos se carga en un registro especial utilizado por el hardware de gestión de la memoria.

Considérese una dirección de $n+m$ bits, donde los n bits de la izquierda corresponden al número de segmento y los m bits de la derecha corresponden al desplazamiento. En el ejemplo (Figura 7.11c), $n = 4$ y $m = 12$. Por tanto, el tamaño de segmento máximo es $2^{12} = 4096$. Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de segmento como los n bits de la izquierda de la dirección lógica.
- Utilizar el número de segmento como un índice a la tabla de segmentos del proceso para encontrar la dirección física inicial del segmento.

- Comparar el desplazamiento, expresado como los *m* bits de la derecha, y la longitud del segmento. Si el desplazamiento es mayor o igual que la longitud, la dirección no es válida.
- La dirección física deseada es la suma de la dirección física inicial y el desplazamiento.

En el ejemplo, se parte de la dirección lógica 0001001011110000, que corresponde al segmento número 1, desplazamiento 752. Supóngase que este segmento reside en memoria principal, comenzando en la dirección física inicial 0010000000100000. Por tanto, la dirección física es 0010000000100000 + 001011110000 = 0010001100010000 (Figura 7.12b).

Resumiendo, con la segmentación simple, un proceso se divide en un conjunto de segmentos que no tienen que ser del mismo tamaño. Cuando un proceso se trae a memoria, todos sus segmentos se cargan en regiones de memoria disponibles, y se crea la tabla de segmentos.

7.5. RESUMEN

Una de las tareas más importantes y complejas de un sistema operativo es la gestión de memoria. La gestión de memoria implica tratar la memoria principal como un recurso que debe asignarse y compartirse entre varios procesos activos. Para utilizar el procesador y las utilidades de E/S eficientemente, es deseable mantener tantos procesos en memoria principal como sea posible. Además, es deseable también liberar a los programadores de tener en cuenta las restricciones de tamaño en el desarrollo de los programas.

Las herramientas básicas de gestión de memoria son la paginación y la segmentación. Con la paginación, cada proceso se divide en un conjunto de páginas de tamaño fijo y de un tamaño relativamente pequeño. La segmentación permite el uso de piezas de tamaño variable. Es también posible combinar la segmentación y la paginación en un único esquema de gestión de memoria.

7.6. LECTURAS RECOMENDADAS

Los libros de sistema operativos recomendados en la Sección 2.9 proporcionan cobertura para la gestión de memoria.

Debido a que el sistema de particionamiento se ha suplantado por técnicas de memoria virtual, la mayoría de los libros sólo cubren superficialmente el tema. Uno de los tratamientos más completos e interesantes se encuentra en [MILE92]. Una discusión más profunda de las estrategias de particionamiento se encuentra en [KNUT97].

Los temas de enlace y carga se cubren en muchos libros de desarrollo de programas, arquitectura de computadores y sistemas operativos. Un tratamiento particularmente detallado es [BECK90]. [CLAR98] también contiene una buena discusión. Una discusión práctica en detalle de este tema, con numerosos ejemplos de sistemas operativos, es [LEVI99].

BECK90 Beck, L. *System Software*. Reading, MA: Addison-Wesley, 1990.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work*. Upper Saddle River, NJ: Prentice Hall, 1998.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1997.

LEVI99 Levine, J. *Linkers and Loaders*. New York: Elsevier Science and Technology, 1999.

MILE92 Milenkovic,M. *Operating Systems: Concepts and Design*. New York: McGraw-Hill, 1992.

7.7. TÉRMINOS CLAVE, CUESTIONES DE REVISIÓN Y PROBLEMAS

TÉRMINOS CLAVE

Carga	Enlace dinámico	Particionamiento
Carga absoluta	Enlazado	Particionamiento dinámico
Carga en tiempo real dinámica	Fragmentación externa	Particionamiento fijo
Carga reubicable	Fragmentación interna	Protección
Compactación	Gestión de memoria	Reubicación
Compartición	Marca	Segmentación
Dirección física	Organización lógica	Sistema XXXX
Dirección lógica	Organización física	Tabla de páginas
Dirección relativa	Página	
Editor de enlaces	Paginación	

CUESTIONES DE REVISIÓN

7.1. ¿Qué requisitos se intenta satisfacer en gestión de la memoria?

7.2. ¿Por qué es deseable la capacidad para reubicar procesos?

7.3. ¿Por qué no es posible forzar la protección de la memoria en tiempo de compilación?

7.4. ¿Qué razones existen para permitir que dos o más procesos accedan a una misma región de la memoria?

7.5. En un esquema de particionamiento fijo, ¿cuáles son las ventajas de utilizar particiones de distinto tamaño?

7.6. ¿Cuál es la diferencia entre fragmentación interna y externa?

7.7. ¿Cuáles son las distinciones entre direcciones lógicas, relativas y físicas?

7.8. ¿Cuál es la diferencia entre una página y un marco?

7.9. ¿Cuál es la diferencia entre una página y un segmento?

PROBLEMAS

7.1. En la Sección 2.3, se listaron cinco objetivos de la gestión de la memoria y en la Sección 7.1 cinco requisitos. Discutir si cada lista incluye los aspectos tratados en la otra lista.

7.2. Considérese un esquema de particionamiento fijo con particiones de igual tamaño de 2^{16} bytes y una memoria principal total de tamaño 2^{24} bytes. Por cada proceso residente, se mantiene una tabla de procesos que incluye un puntero a una partición. ¿Cuántos bits necesita el puntero?

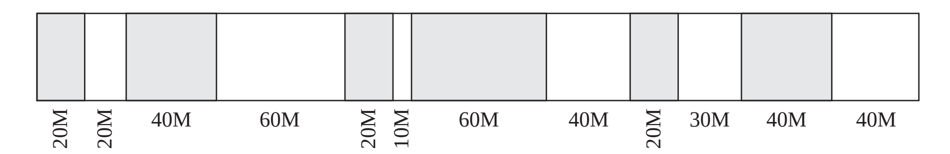
7.3. Considérese un esquema de particionamiento dinámico. Demostrar que, en media, la memoria contiene la mitad de huecos que de segmentos.

7.4. Para implementar los diferentes algoritmos de colocación discutidos para el particionamiento dinámico (Sección 7.2), se debe guardar una lista de los bloques libres de memoria.

Para cada uno de los tres métodos discutidos (mejor ajuste (*best-fit*), primer ajuste (*first-fit*) y próximo ajuste (*next-fit*)), ¿cuál es la longitud media de la búsqueda?

7.5. Otro algoritmo de colocación para el particionamiento dinámico es el de peor ajuste (*worst-fit*). En este caso, se utiliza el mayor bloque de memoria libre para un proceso. Discutir las ventajas e inconvenientes de este método comparado con el primer, próximo y mejor ajuste. ¿Cuál es la longitud media de la búsqueda para el peor ajuste?

7.6. Si se utiliza un esquema de particionamiento dinámico y en un determinado momento la configuración de memoria es la siguiente:



Las áreas sombreadas son bloques asignados; las áreas blancas son bloques libres. Las siguientes tres peticiones de memoria son de 40M, 20M y 10M. Indíquese la dirección inicial para cada uno de los tres bloques utilizando los siguientes algoritmos de colocación:

a) Primer ajuste

b) Mejor ajuste

c) Siguiente ajuste. Asúmase que el bloque añadido más recientemente se encuentra al comienzo de la memoria.

d) Peor ajuste

7.7. Un bloque de memoria de 1 Mbyte se asigna utilizando el sistema *buddy*:

a) Mostrar los resultados de la siguiente secuencia en una figura similar a la Figura 7.6: Petición 70; Petición 35; Petición 80; Respuesta A; Petición 60; Respuesta B; Respuesta D; Respuesta C.

b) Mostrar la representación de árbol binario que sigue a Respuesta B.

7.8. Considérese un sistema *buddy* en el que un determinado bloque en la asignación actual tiene la dirección 011011110000.

a) Si el bloque es de tamaño 4, ¿cuál es la dirección binaria de su bloque compañero o *buddy*?

b) Si el bloque es de tamaño 16, ¿cuál es la dirección binaria de su bloque compañero o *buddy*?

7.9. Sea $buddy_k(x)$ = dirección del bloque de tamaño 2^k, cuya dirección es x. Escribir una expresión general para el bloque compañero *buddy de* $buddy_k(x)$.

7.10. La secuencia Fibonacci se define como sigue:

$$F_o=0,\ F_1=1,\ F_{n+2}= F_{n+1} + F_n,\ n \geq 0$$

a) ¿Podría utilizarse esta secuencia para establecer un sistema *buddy*?

b) ¿Cuál sería la ventaja de este sistema respecto al sistema *buddy* binario descrito en este capítulo?

7.11. Durante el curso de ejecución de un programa, el procesador incrementará en una palabra los contenidos del registro de instrucciones (contador de programa) después de que se cargue cada instrucción, pero alterará los contenidos de dicho registro si encuentra un salto o instrucción de llamada que provoque la ejecución de otra parte del programa. Ahora considérese la Figura 7.8. Hay dos alternativas respecto a las direcciones de la instrucción:

- Mantener una dirección relativa en el registro de instrucciones y hacer la traducción de direcciones dinámica utilizando el registro de instrucciones como entrada. Cuando se encuentra un salto o una llamada, la dirección relativa generada por dicho salto o llamada se carga en el registro de instrucciones.
- Mantener una dirección absoluta en el registro de instrucciones. Cuando se encuentra un salto o una llamada, se emplea la traducción de direcciones dinámica, almacenando los resultados en el registro de instrucciones.

¿Qué opción es preferible?

7.12. Considérese un sistema de paginación sencillo con los siguientes parámetros: 2^{32} bytes de memoria física; tamaño de página de 2^{10} bytes; 2^{16} páginas de espacio de direccionamiento lógico.

a) ¿Cuántos bits hay en una dirección lógica?

b) ¿Cuántos bytes hay en un marco?

c) ¿Cuántos bits en la dirección física especifica el marco?

d) ¿Cuántas entradas hay en la tabla de páginas?

e) ¿Cuántos bits hay en cada entrada de la tabla de páginas? Asúmase que cada entrada de la tabla de páginas incluye un bit válido/inválido.

7.13. Una dirección virtual *a* en un sistema de paginación es equivalente a un par *(p,w)*, en el cual *p* es un número de pagina y *w* es un número de bytes dentro de la página. Sea *z* el número de bytes de una página. Encontrar ecuaciones algebraicas que muestren *p* y *w* como funciones de *z* y *a*.

7.14. Considérese un sistema de segmentación sencillo que tiene la siguiente tabla de segmentos:

Dirección inicial	Longitud (bytes)
660	248
1752	422
222	198
996	604

Para cada una de las siguientes direcciones lógicas, determina la dirección física o indica si se produce un fallo de segmento:

a) 0,198

b) 2,156

c) 1,530

d) 3,444

e) 0,222

7.15. Considérese una memoria en la cual se colocan segmentos contiguos S_1, S_2, …, S_n en su orden de creación, desde un extremo del dispositivo al otro, como se sugiere en la siguiente figura:

S_1	S_2	• • •	S_n	Hueco

Cuando se crea el segmento contiguos S_{n+1},se coloca inmediatamente después de S_n, incluso si algunos de los segmentos S_1, S_2, …, S_n ya se hubieran borrado. Cuando el límite entre segmentos (en uso o borrados) y el hueco alcanzan el otro extremo de memoria, los segmentos en uso se compactan.

a) Mostrar que la fracción de tiempo *F* utilizada para la compactación cumple la siguiente inigualdad:

$$F \geq \frac{1-f}{1+kf}, \text{ donde } k = \frac{t}{2s} - 1$$

donde

s = longitud media de un segmento, en palabras

t = tiempo de vida medio de un segmento, en referencias a memoria

f = fracción de la memoria que no se utiliza bajo condiciones de equilibrio

Sugerencia: Encontrar la velocidad media a la que los límites cruzan la memoria y

b) Encontrar *F* para *f*=0,2, *t*=1000 y s=50.

APÉNDICE 7A CARGA Y ENLACE

El primer paso en la creación de un proceso activo es cargar un programa en memoria principal y crear una imagen del proceso (Figura 7.13). Figura 7.14 muestra un escenario típico para la mayoría de los sistemas. La aplicación está formada por varios módulos compilados o ensamblados en formato de código objeto. Éstos son enlazados para resolver todas las referencias entre los módulos. Al mismo tiempo, se resuelven las referencias a rutinas de biblioteca. Las rutinas de biblioteca pueden incorporarse al programa o hacerle referencia como código compartido que el sistema operativo proporciona en tiempo de ejecución. En este apéndice, se resumen las características clave de los enlazadores y cargadores. Por motivos de claridad en la presentación, se comienza con una descripción de la tarea de carga cuando sólo se tiene un módulo de programa; en este caso no se requiere enlace.

CARGA

En la Figura 7.14, el cargador coloca el módulo de carga en la memoria principal, comenzando en la ubicación *x*. En la carga del programa, se debe satisfacer el requisito de direccionamiento mostrado en la Figura 7.1. En general, se pueden seguir tres técnicas diferentes:

- Carga absoluta
- Carga reubicable
- Carga dinámica en tiempo real

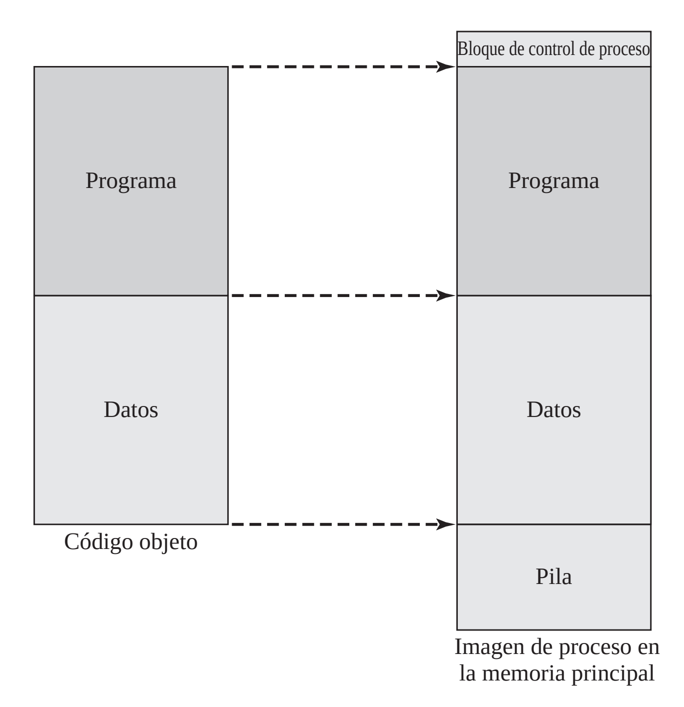


Figura 7.13. La función de carga.

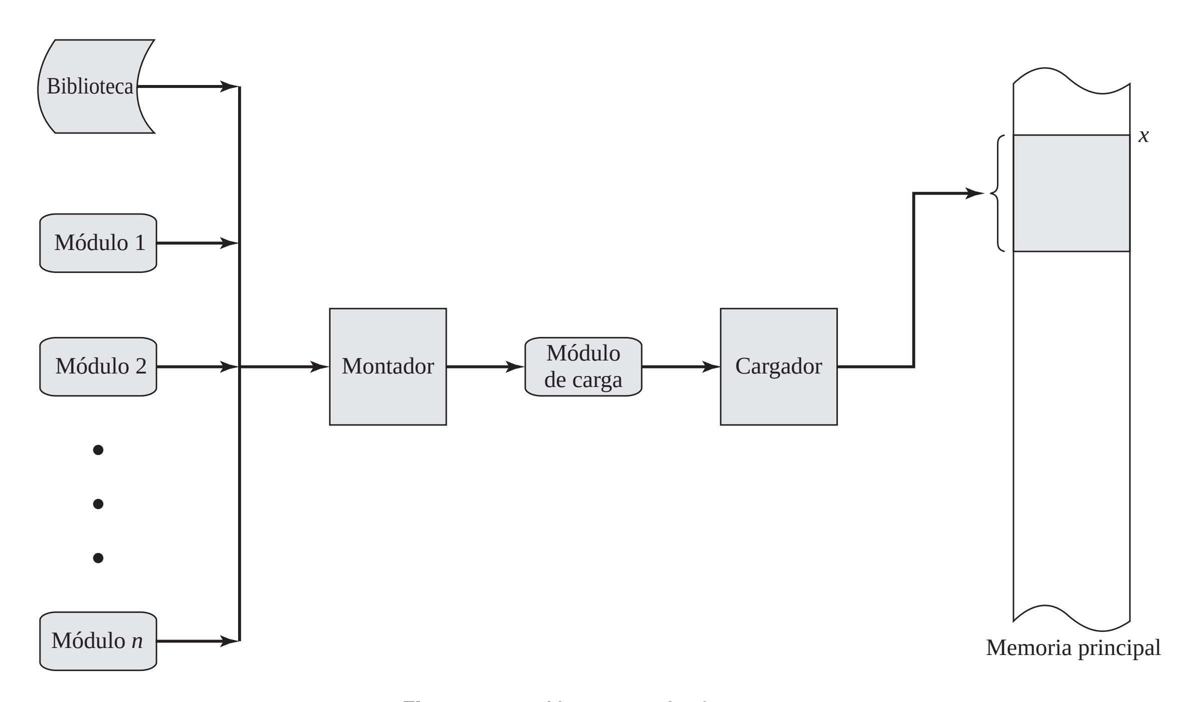


Figura 7.14. Un escenario de carga.

Carga absoluta

Un cargador absoluto requiere que un módulo de carga dado debe cargarse siempre en la misma ubicación de la memoria principal. Por tanto, en el módulo de carga presentado al cargador, todas las referencias a direcciones deben ser direcciones de memoria principal específicas o absolutas. Por ejemplo, si en la Figura 7.14 *x* es la ubicación 1024, entonces la primera palabra de un módulo de carga destinado para dicha región de memoria, tiene la dirección 1024.

La asignación de valores de direcciones específicas a referencias de programa dentro de un programa lo puede hacer el programador o se hacen en tiempo de compilación o ensamblado (Tabla 7.2a). La primera opción tiene varias desventajas. Primero, cada programador debe conocer la estrategia de asignación para colocar los módulos en memoria principal. Segundo, si se hace cualquier modificación al programa que implique inserciones o borrados en el cuerpo del módulo, entonces todas las direcciones deben alterarse. Por tanto, es preferible permitir que las referencias de memoria dentro de los programas se expresen simbólicamente, y entonces resolver dichas referencias simbólicas en tiempo de compilación o ensamblado. Esto queda reflejado en la Figura 7.15. Cada referencia a una instrucción o elemento de datos se representa inicialmente como un símbolo. A la hora de preparar el módulo para su entrada a un cargador absoluto, el ensamblador o compilador convertirá todas estas referencias a direcciones específicas (en este ejemplo, el módulo se carga en la dirección inicial de 1024), tal como se muestra en la Figura 7.15b.

Tabla 7.2. Asociación de direcciones.

(a) Cargador

Tiempo de asociación	Función
Tiempo de programación	El programador especifica directamente en el propio programa todas las direcciones físicas reales.
Tiempo de compilación o ensamblado	El programa contiene referencias a direcciones simbólicas y el compilador o ensamblador las convierte a direcciones físicas reales.
Tiempo de carga	El compilador o ensamblador produce direcciones relativas. El cargador las traduce a direcciones absolutas cuando se carga el programa.
Tiempo de ejecución	El programa cargador retiene direcciones relativas. El hardware del procesador las convierte dinámicamente a direcciones absolutas.

(b) Montador

Tiempo de montaje	Función
Tiempo de programación	No se permiten referencias a programas o datos externos. El programador debe colocar en el programa el código fuente de todos los subprogramas que invoque.
Tiempo de compilación o ensamblado	El ensamblador debe traer el código fuente de cada subrutina que se referencia y ensamblarlo como una unidad.
Creación de módulo de carga	Todos los módulos objeto se han ensamblado utilizando direcciones relativas. Estos módulos se enlazan juntos y todas las referencias se restablecen en relación al origen del módulo de carga final.
Tiempo de carga	Las referencias externas no se resuelven hasta que el módulo de carga se carga en memoria principal. En ese momento, los módulos con enlace dinámico referenciados se adjuntan al módulo de carga y el paquete completo se carga en memoria principal o virtual.
Tiempo de ejecución	Las referencias externas no se resuelven hasta que el procesador ejecuta la llamada externa. En ese momento, el proceso se interrumpe y el módulo deseado se enlaza al programa que lo invoca.

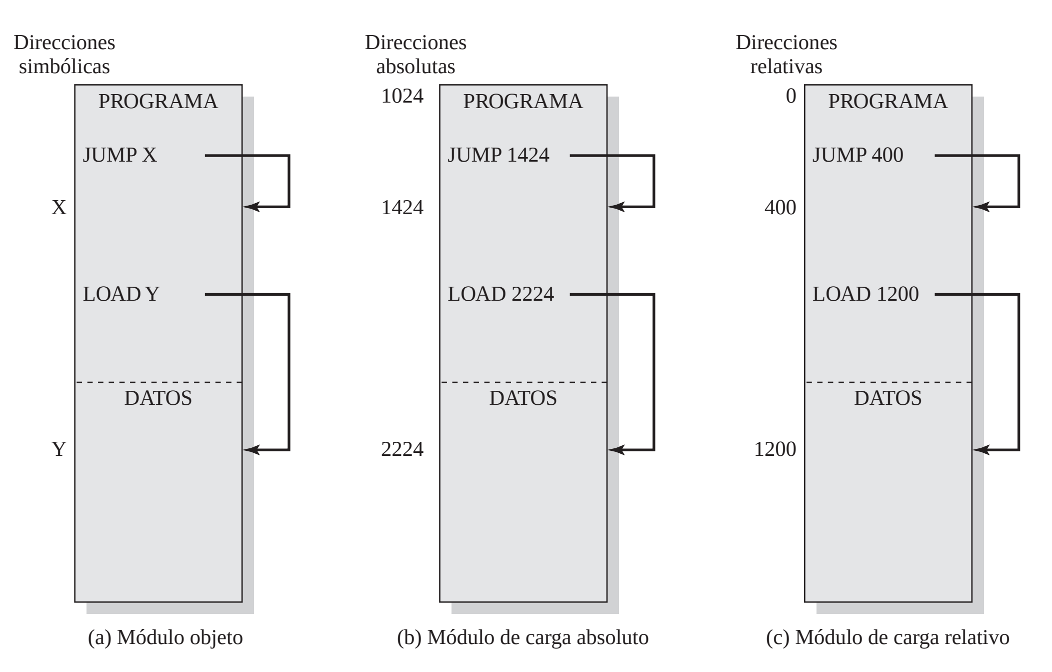


Figura 7.15. Módulos de carga absolutos y reubicables.

Carga reubicable

La desventaja de enlazar referencias de memoria a direcciones específicas antes de la carga es que el módulo de carga resultante sólo se puede colocar en una región específica de memoria principal. Sin embargo, cuando muchos programas comparten memoria principal, podría no ser deseable decidir al principio en qué región de la memoria se debe cargar un módulo particular. Es mejor tomar esta decisión en tiempo de carga. Por tanto, necesitamos un módulo de carga que pueda ubicarse en cualquier lugar de la memoria principal.

Para satisfacer este nuevo requisito, el ensamblador o compilador no produce direcciones de memoria reales (direcciones absolutas), sino direcciones relativas a algún punto conocido, tal como el inicio del programa. Esta técnica se muestra en la Figura 7.15c. El comienzo del módulo de carga se asigna a la dirección relativa 0, y el resto de las referencias de memoria dentro del módulo se expresan relativas al comienzo del módulo.

Con todas las referencias de la memoria expresadas en formato relativo, colocar el módulo en el lugar adecuado se convierte en una tarea simple para el cargador. Si el módulo se carga al comienzo de la ubicación *x*, entonces el cargador debe simplemente añadir *x* a cada referencia de la memoria cuando carga el módulo en memoria. Para asistir en esta tarea, el módulo cargado debe incluir información que dice el cargador donde están las referencias de memoria y cómo se van a interpretar (normalmente relativo al origen del programa, pero también es posible relativo a algún otro punto del programa, tal como la ubicación actual). El compilador o ensamblador prepara este conjunto de información, lo que se denomina normalmente diccionario de reubicación.

Carga dinámica en tiempo real

Los cargadores reubicables son comunes y proporcionan beneficios obvios si se comparan con los cargadores absolutos. Sin embargo, en un entorno de multiprogramación, incluso en uno que no de-

penda de la memoria virtual, el esquema de carga reubicable no es adecuado. A lo largo del libro, nos hemos referido a la necesidad de traer y quitar imágenes de procesos de la memoria principal a fin de maximizar la utilización del procesador. Para maximizar la utilización de la memoria principal, sería importante poder intercambiar las imágenes de los procesos en diferentes localizaciones en diferentes momentos. Por tanto, un programa, una vez cargado, puede intercambiarse a disco y a memoria en diferentes ubicaciones. Esto sería imposible si las referencias de la memoria se limitan a direcciones absolutas en tiempo de carga inicial.

La alternativa es posponer el cálculo de una dirección absoluta hasta que se necesite realmente en tiempo de ejecución. Para este propósito, el módulo de carga se carga en la memoria principal con todas las referencias de la memoria en formato relativo (Figura 7.15c). Hasta que una instrucción no se ejecuta realmente, no se calcula la dirección absoluta. Para asegurar que esta función no degrada el rendimiento, la realiza el hardware de procesador especial en lugar de llevarse a cabo por software. El hardware se describe en la Sección 7.2.

El cálculo dinámico de direcciones proporciona una flexibilidad total. Un programa se carga en cualquier región de la memoria principal. A continuación, la ejecución del programa se puede interrumpir y el programa se puede intercambiar entre disco y memoria, para posteriormente intercambiarse en una localización diferente.

ENLACE

La función de un montador es tomar como entrada una colección de módulos objeto y producir un módulo de carga, formado por un conjunto integrado de programa y módulos de datos, que se pasará al cargador. En cada módulo objeto, podría haber referencias a direcciones de otros módulos. Cada una de estas referencias sólo se puede expresar simbólicamente en un módulo objeto no enlazado. El montador crea un único módulo de carga que se une de forma contigua a todos los módulos objeto. Cada referencia entre módulos debe cambiarse: una dirección simbólica debe convertirse en una referencia a una ubicación dentro del módulo de carga. Por ejemplo, el módulo A en la Figura 7.16a contiene una invocación a un procedimiento del módulo B. Cuando estos módulos se combinan en el módulo de carga, esta referencia simbólica al módulo B se cambia por una referencia específica a la localización del punto de entrada de B dentro del módulo de carga.

Editor de enlace

La naturaleza de este enlace de direcciones dependerá del tipo de módulo de carga que se cree y cuando se lleve a cabo el proceso de enlace (Tabla 7.2b). Si se desea un módulo de carga reubicable, como suele ser lo habitual, el enlace se hace normalmente de la siguiente forma. Cada módulo objeto compilado o ensamblado se crea con referencias relativas al comienzo del módulo objeto. Todos estos módulos se colocan juntos en un único módulo de carga reubicable con todas las referencias relativas al origen del módulo de carga. Este módulo se puede utilizar como entrada para la carga reubicable o carga dinámica en tiempo de ejecución.

Un montador que produce un módulo de carga reubicable se denomina frecuentemente editor de enlace. La Figura 7.16 ilustra la función del editor de enlace.

Montador dinámico

Al igual que con la carga, también es posible posponer algunas funciones relativas al enlace. El término *enlace dinámico* se utiliza para denominar la práctica de posponer el enlace de algunos módulos

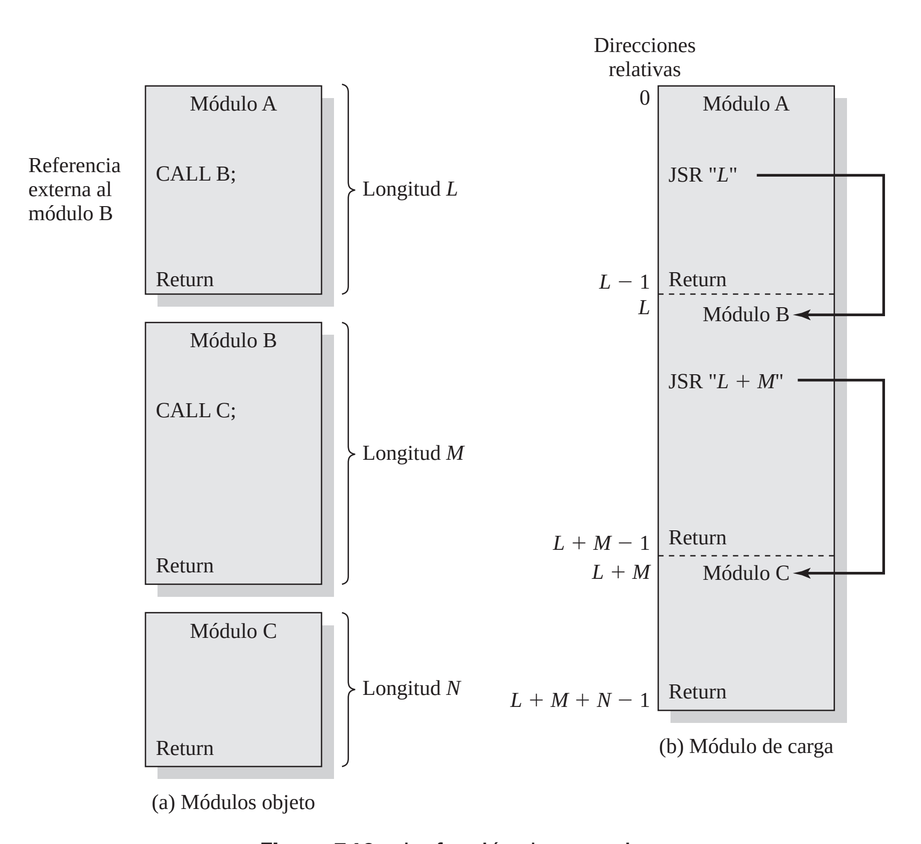


Figura 7.16. La función de montaje.

externos hasta después de que el módulo de carga se cree. Por tanto, el módulo de carga contiene referencias sin resolver a otros programas. Estas referencias se pueden resolver o bien en tiempo de carga o bien en tiempo de ejecución.

Para el **enlace dinámico en tiempo de carga**, se deben seguir los siguientes pasos. El módulo de carga (módulo de aplicación) debe llevarse a memoria para cargarlo. Cualquier referencia a un módulo externo (módulo destino) provoca que el cargador encuentre al módulo destino, lo cargue y altere la referencia a una dirección relativa a la memoria desde el comienzo del módulo de aplicación. Hay varias ventajas de esta técnica frente al enlace estático:

- Se facilita incorporar versiones modificadas o actualizadas del módulo destino, el cual puede ser una utilidad del sistema operativo o algunas otras rutinas de propósito general. Con el enlace estático, un cambio a ese módulo requeriría el reenlace del módulo de aplicación completo. No sólo es ineficiente, sino que puede ser imposible en algunas circunstancias. Por ejemplo, en el campo de los ordenadores personales, la mayoría del software comercial es entregado en el formato del módulo de carga; no se entregan las versiones fuente y objeto.

- Tener el código destino en un fichero con enlace dinámico facilita el camino para compartir código de forma automática. El sistema operativo puede reconocer que más de una aplicación utiliza el mismo código destino porque carga y enlaza dicho código. Puede utilizar esta información para cargar una única copia del código destino y enlazarlo a ambas aplicaciones, en lugar de tener que cargar una copia para cada aplicación.

- Facilita a los desarrolladores de software independientes extender la funcionalidad de un sistema operativo ampliamente utilizado, tal como Linux. Un desarrollador puede implementar una nueva función que puede ser útil a una variedad de aplicaciones y empaquetarla como un módulo de enlace dinámico.

Con **enlace dinámico en tiempo de ejecución**, algunos de los enlaces son pospuestos hasta el tiempo de ejecución. Las referencias externas a los módulos destino quedan en el programa cargado. Cuando se realiza una llamada a un módulo ausente, el sistema operativo localiza el módulo, lo carga y lo enlaza al módulo llamante.

Se ha visto que la carga dinámica permite que se pueda mover un módulo de carga entero; sin embargo, la estructura del módulo es estática, permaneciendo sin cambios durante la ejecución del proceso y de una ejecución a otra. Sin embargo, en algunos casos, no es posible determinar antes de la ejecución qué módulos objeto se necesitarán. Esta situación es tipificada por las aplicaciones de procesamiento de transacciones, como las de un sistema de reserva de vuelos o una aplicación bancaria. La naturaleza de la transacción especifica qué módulos de programa se requieren, y éstos son cargados y enlazados con el programa principal. La ventaja del uso de un montador dinámico es que no es necesario asignar memoria a unidades del programa a menos que dichas unidades se referencien. Esta capacidad se utiliza para dar soporte a sistemas de segmentación.

Una mejora adicional es posible: una aplicación no necesita conocer los nombres de todos los módulos o puntos de entrada que pueden llamarse. Por ejemplo, puede escribirse un programa de dibujo para trabajar con una gran variedad de trazadores, cada uno de los cuales se gestiona por un controlador diferente. La aplicación puede aprender el nombre del trazador que está actualmente instalado en el sistema por otro proceso o buscando en un fichero de configuración. Esto permite al usuario de la aplicación instalar un nuevo trazador que no exista en el tiempo en que la aplicación se escribió.

CAPÍTULO 8

Memoria virtual

◆ ◆

En el Capítulo 7 se vieron los conceptos de paginación y segmentación y se analizaron sus limitaciones. Ahora vamos a entrar a discutir el concepto de memoria virtual. Un análisis de este concepto es complicado por el hecho de que la gestión de la memoria es una interacción compleja entre el hardware del procesador y el sistema operativo. Nos centraremos primero en los aspectos hardware de la memoria virtual, observando el uso de la paginación, segmentación, y combinación de paginación y segmentación. Después veremos los aspectos relacionados con el diseño de los servicios de la memoria virtual en el sistema operativo.

◆ ◆

8.1. HARDWARE Y ESTRUCTURAS DE CONTROL

Comparando la paginación sencilla y la segmentación sencilla, por un lado, tenemos una distinción entre particionamiento estático y dinámico, y por otro, tenemos los fundamentos de comienzo de la gestión de la memoria. Las dos características de la paginación y la segmentación que son la clave de este comienzo son:

1. Todas las referencias a la memoria dentro un proceso se realizan a direcciones lógicas, que se traducen dinámicamente en direcciones físicas durante la ejecución. Esto significa que un proceso puede ser llevado y traído a memoria de forma que ocupe diferentes regiones de la memoria principal en distintos instantes de tiempo durante su ejecución.
2. Un proceso puede dividirse en varias porciones (páginas o segmentos) y estas porciones no tienen que estar localizadas en la memoria de forma contigua durante la ejecución. La combinación de la traducción de direcciones dinámicas en ejecución y el uso de una tabla de páginas o segmentos lo permite.

Ahora veamos cómo comenzar con la memoria dinámica. *Si las dos características anteriores se dan, entonces es necesario que todas las páginas o todos los segmentos de un proceso se encuentren en la memoria principal durante la ejecución*. Si la porción (segmento o página) en la que se encuentra la siguiente instrucción a buscar está y si la porción donde se encuentra la siguiente dirección de datos que se va a acceder también está, entonces al menos la siguiente instrucción se podrá ejecutar.

Consideremos ahora cómo se puede realizar esto. De momento, vamos a hablar en términos generales, y usaremos el término *porción* para referirnos o bien a una página o un segmento, dependiendo si estamos empleando paginación o segmentación. Supongamos que se tiene que traer un nuevo proceso de memoria. El sistema operativo comienza trayendo únicamente una o dos porciones, que incluye la porción inicial del programa y la porción inicial de datos sobre la cual acceden las primeras instrucciones acceden. Esta parte del proceso que se encuentra realmente en la memoria principal para, cualquier instante de tiempo, se denomina **conjunto residente** del proceso. Cuando el proceso está ejecutándose, las cosas ocurren de forma suave mientras que todas las referencias a la memoria se encuentren dentro del conjunto residente. Usando una tabla de segmentos o páginas, el procesador siempre es capaz de determinar si esto es así o no. Si el procesador encuentra una dirección lógica que no se encuentra en la memoria principal, generará una interrupción indicando un fallo de acceso a la memoria. El sistema operativo coloca al proceso interrumpido en un estado de bloqueado y toma el control. Para que la ejecución de este proceso pueda reanudarse más adelante, el sistema operativo necesita traer a la memoria principal la porción del proceso que contiene la dirección lógica que ha causado el fallo de acceso. Con este fin, el sistema operativo realiza una petición de E/S, una lectura a disco. Después de realizar la petición de E/S, el sistema operativo puede activar otro proceso que se ejecute mientras el disco realiza la operación de E/S. Una vez que la porción solicitada se ha traído a

la memoria principal, una nueva interrupción de E/S se lanza, dando control de nuevo al sistema operativo, que coloca al proceso afectado de nuevo en el estado Listo.

Al lector se le puede ocurrir cuestionar la eficiencia de esta maniobra, en la cual a un proceso que se puede estar ejecutando resulta necesario interrumpirlo sin otro motivo que el hecho de que no se ha llegado a cargar todas las porciones necesarias de dicho proceso. De momento, vamos a posponer esta cuestión con la garantía de que la eficiencia es verdaderamente posible. En su lugar, vamos a ponderar las implicaciones de nuestra nueva estrategia. Existen dos implicaciones, la segunda más sorprendente que la primera, y ambas dirigidas a mejorar la utilización del sistema:

1. **Pueden mantenerse un mayor número de procesos en memoria principal**. Debido a que sólo vamos a cargar algunas de las porciones de los procesos a ejecutar, existe espacio para más procesos. Esto nos lleva a una utilización más eficiente del procesador porque es más probable que haya al menos uno o más de los numerosos procesos que se encuentre en el estado Listo, en un instante de tiempo concreto.
2. **Un proceso puede ser mayor que toda la memoria principal**. Se puede superar una de las restricciones fundamentales de la programación. Sin el esquema que hemos estado discutiendo, un programador debe estar realmente atento a cuánta memoria está disponible. Si el programa que está escribiendo es demasiado grande, el programador debe buscar el modo de estructurar el programa en fragmentos que pueden cargarse de forma separada con un tipo de estrategia de superposición (*overlay*). Con la memoria virtual basada en paginación o segmentación, este trabajo se delega al sistema operativo y al hardware. En lo que concierne al programador, él está trabajando con una memoria enorme, con un tamaño asociado al almacenamiento en disco. El sistema operativo automáticamente carga porciones de un proceso en la memoria principal cuando éstas se necesitan.

Debido a que un proceso ejecuta sólo en la memoria principal, esta memoria se denomina **memoria real**. Pero el programador o el usuario perciben una memoria potencialmente mucho más grande —la cual se encuentra localizada en disco. Esta última se denomina **memoria virtual**. La memoria virtual permite una multiprogramación muy efectiva que libera al usuario de las restricciones excesivamente fuertes de la memoria principal. La Tabla 8.1 recoge las características de la paginación y la segmentación, con y sin el uso de la memoria virtual.

PROXIMIDAD Y MEMORIA VIRTUAL

Los beneficios de la memoria virtual son atractivos, ¿pero el esquema es verdaderamente práctico? En su momento, hubo un importante debate sobre este punto, pero la experiencia de numerosos sistemas operativos ha demostrado, más allá de toda duda, que la memoria virtual realmente funciona. La memoria virtual, basada en paginación o paginación más segmentación, se ha convertido, en la actualidad, en una componente esencial de todos los sistemas operativos contemporáneos.

Para entender cuál es el aspecto clave, y por qué la memoria virtual era la causa de dicho debate, examinemos de nuevo las tareas del sistema operativo relacionadas con la memoria virtual. Se va a considerar un proceso de gran tamaño, consistente en un programa largo más un gran número de vectores de datos. A lo largo de un corto periodo de tiempo, la ejecución se puede acotar a una pequeña sección del programa (por ejemplo, una subrutina) y el acceso a uno o dos vectores de datos únicamente. Si es así, sería verdaderamente un desperdicio cargar docenas de porciones de dicho proceso cuando sólo unas pocas porciones se usarán antes de que el programa se suspenda o se mande a zona de intercambio o *swap*. Se puede hacer un mejor uso de la memoria cargando únicamente unas pocas porciones. Entonces, si el programa salta a una destrucción o hace referencia a un dato que se en-

Tabla 8.1. Características de la paginación y la segmentación.

Paginación sencilla	Paginación con memoria virtual	Segmentación sencilla	Segmentación con memoria virtual
Memoria principal particionada en fragmentos pequeños de un tamaño fijo llamados marcos	Memoria principal particionada en fragmentos pequeños de un tamaño fijo llamados marcos	Memoria principal no particionada	Memoria principal no particionada
Programa dividido en páginas por el compilador o el sistema de gestión de la memoria	Programa dividido en páginas por el compilador o el sistema de gestión de la memoria	Los segmentos de programa se especifican por el programador al compilador (por ejemplo, la decisión se toma por parte el programador)	Los segmentos de programa se especifican por el programador al compilador (por ejemplo, la decisión se toma por parte el programador)
Fragmentación interna dentro de los marcos	Fragmentación interna dentro de los marcos	Sin fragmentación interna	Sin fragmentación interna
Sin fragmentación externa	Sin fragmentación externa	Fragmentación externa	Fragmentación externa
El sistema operativo debe mantener una tabla de páginas por cada proceso mostrando en el marco que se encuentra cada página ocupada	El sistema operativo debe mantener una tabla de páginas por cada proceso mostrando en el marco que se encuentra cada página ocupada	El sistema operativo debe mantener una tabla de segmentos por cada proceso mostrando la dirección de carga y la longitud de cada segmento	El sistema operativo debe mantener una tabla de segmentos por cada proceso mostrando la dirección de carga y la longitud de cada segmento
El sistema operativo debe mantener una lista de marcos libres	El sistema operativo debe mantener una lista de marcos libres	El sistema operativo debe mantener una lista de huecos en la memoria principal	El sistema operativo debe mantener una lista de huecos en la memoria principal
El procesador utiliza el número de página, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de página, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de segmento, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de segmento, desplazamiento para calcular direcciones absolutas
Todas las páginas del proceso deben encontrarse en la memoria principal para que el proceso se pueda ejecutar, salvo que se utilicen solapamientos (*overlays*)	No se necesita mantener todas las páginas del proceso en los marcos de la memoria principal para que el proceso se ejecute. Las páginas se pueden leer bajo demanda	Todos los segmentos del proceso deben encontrarse en la memoria principal para que el proceso se pueda ejecutar, salvo que se utilicen solapamientos (*overlays*)	No se necesitan mantener todos los segmentos del proceso en la memoria principal para que el proceso se ejecute. Los segmentos se pueden leer bajo demanda
	La lectura de una página a memoria principal puede requerir la escritura de una página a disco		La lectura de un segmento a memoria principal puede requerir la escritura de uno o más segmentos a disco

cuentra en una porción de memoria que no está en la memoria principal, entonces se dispara un fallo. Éste indica al sistema operativo que debe conseguir la porción deseada.

Así, en cualquier momento, sólo unas pocas porciones de cada proceso se encuentran en memoria, y por tanto se pueden mantener más procesos alojados en la misma. Además, se ahorra tiempo porque las porciones del proceso no usadas no se expulsarán de la memoria a *swap* y de *swap* a la memoria. Sin embargo, el sistema operativo debe ser inteligente a la hora de manejar este esquema. En estado estable, prácticamente toda la memoria principal se encontrará ocupada con porciones de procesos, de forma que el procesador y el sistema operativo tengan acceso directo al mayor número posible de procesos. Así, cuando el sistema operativo traiga una porción a la memoria, debe expulsar otra. Si elimina una porción justo antes de que vaya a ser utilizada, deberá recuperar dicha porción de nuevo casi de forma inmediata. Un abuso de esto lleva a una condición denominada **trasiego (*thrashing*)**: el sistema consume la mayor parte del tiempo enviando y trayendo porciones de *swap* en lugar de ejecutar instrucciones. Evitar el trasiego fue una de las áreas de investigación principales en la época de los años 70 que condujo una gran variedad de algoritmos complejos pero muy efectivos. En esencia, el sistema operativo trata de adivinar, en base a la historia reciente, qué porciones son menos probables de ser utilizadas en un futuro cercano.

Este razonamiento se basa en la creencia del **principio de proximidad**, que se presentó en el Capítulo 1 (*véase* especialmente el Apéndice 1A). Para resumir, el principio de proximidad indica que las referencias al programa y a los datos dentro de un proceso tienden a agruparse. Por tanto, se resume que sólo unas pocas porciones del proceso se necesitarán a lo largo de un periodo de tiempo corto. También, es posible hacer suposiciones inteligentes sobre cuáles son las porciones del proceso que se necesitarán en un futuro próximo, para evitar este trasiego.

Una forma de confirmar el principio de proximidad es observar el rendimiento de los procesos en un entorno de memoria virtual. En la Figura 8.1 se muestra un famoso diagrama que ilustra de forma clara los principios de proximidad [HATF 72]. Nótese que, durante el tiempo de vida de un proceso las referencias se encuentran acotadas a un subconjunto de sus páginas.

Así pues, vemos que el principio de proximidad sugiere que el esquema de memoria virtual debe funcionar. Para que la memoria virtual resulte práctica y efectiva, se necesitan dos ingredientes. Primero, debe existir un soporte hardware para el esquema de paginación y/o segmentación. Segundo, el sistema operativo debe incluir código para gestionar el movimiento de páginas y/o segmentos entre la memoria secundaria y la memoria principal. En esta sección, examinaremos los aspectos hardware y veremos cuáles son las estructuras de control necesarias, que se crearán y mantendrán por parte del sistema operativo pero que son usadas por el hardware de gestión de la memoria. Se examinarán los aspectos correspondientes al sistema operativo en la siguiente sección.

PAGINACIÓN

El término *memoria virtual* se asocia habitualmente con sistemas que emplean paginación, a pesar de que la memoria virtual basada en segmentación también se utiliza y será tratada más adelante. El uso de paginación para conseguir memoria virtual fue utilizado por primera vez en el computador Atlas [KILB62] y pronto se convirtió en una estrategia usada en general de forma comercial.

En la presentación de la paginación sencilla, indicamos que cada proceso dispone de su propia tabla de páginas, y que todas las páginas se encuentran localizadas en la memoria principal. Cada entrada en la tabla de páginas consiste en un número de marco de la correspondiente página en la memoria principal. Para la memoria virtual basada en el esquema de paginación también se necesita una tabla de páginas. De nuevo, normalmente se asocia una única tabla de páginas a cada proceso. En este caso, sin embargo, las entradas de la tabla de páginas son más complejas (Figura 8.2a). Debido a que

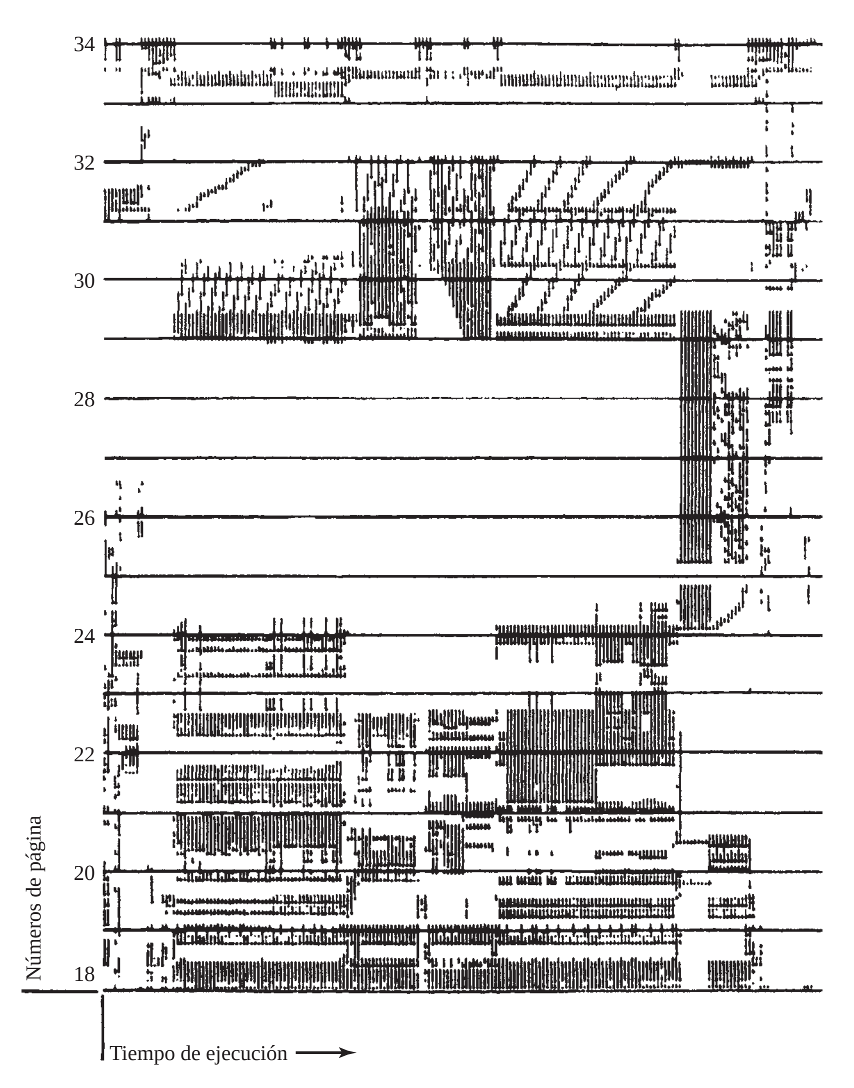


Figura 8.1. Comportamiento de la paginación.

sólo algunas de las páginas de proceso se encuentran en la memoria principal, se necesita que cada entrada de la tabla de páginas indique si la correspondiente página está presente (P) en memoria principal o no. Si el bit indica que la página está en memoria, la entrada también debe indicar el número de marco de dicha página.

La entrada de la tabla de páginas incluye un bit de modificado (M), que indica si los contenidos de la correspondiente página han sido alterados desde que la página se cargó por última vez en la memoria principal. Si no había ningún cambio, no es necesario escribir la página cuando llegue el momento de reemplazarla por otra página en el marco de página que actualmente ocupa. Pueden existir también otros bits de control en estas entradas. Por ejemplo, si la protección y compartición se gestiona a nivel de página, se necesitarán también los bits para este propósito.

Estructura de la tabla de páginas. El mecanismo básico de lectura de una palabra de la memoria implica la traducción de la dirección virtual, o lógica, consistente en un número de página y un des-

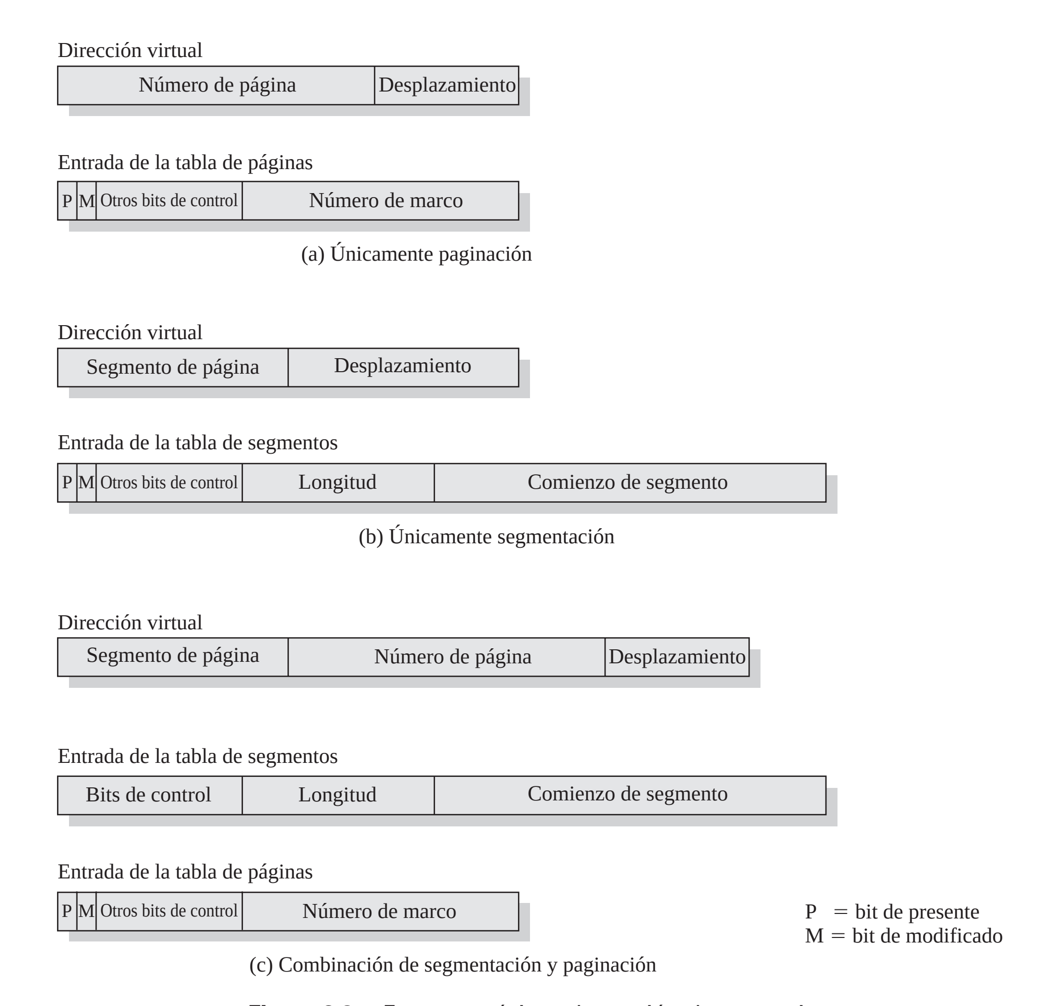


Figura 8.2. Formatos típicos de gestión de memoria.

plazamiento, a la dirección física, consistente en un número de marco y un desplazamiento, usando para ello la tabla de páginas. Debido a que la tabla de páginas es de longitud variable dependiendo del tamaño del proceso, no podemos suponer que se encuentra almacenada en los registros. En lugar de eso, debe encontrarse en la memoria principal para poder ser accedida. La Figura 8.3 sugiere una implementación hardware. Cuando un proceso en particular se encuentra ejecutando, un registro contiene la dirección de comienzo de la tabla de páginas para dicho proceso. El número de página de la dirección virtual se utiliza para indexar esa tabla y buscar el correspondiente marco de página. Éste, combinado con la parte de desplazamiento de la dirección virtual genera la dirección real deseada. Normalmente, el campo correspondiente al número de página es mayor que el campo correspondiente al número de marco de página ($n > m$).

En la mayoría de sistemas, existe una única tabla de página por proceso. Pero cada proceso puede ocupar una gran cantidad de memoria virtual. Por ejemplo, en la arquitectura VAX, cada proceso puede tener hasta 2^{31} = 2 Gbytes de memoria virtual. Usando páginas de 2^9 = 512 bytes, que representa un total de 2^{22} entradas de tabla de página *por cada proceso*. Evidentemente, la cantidad de memoria demandada por las tablas de página únicamente puede ser inaceptablemente grande. Para resolver este problema, la mayoría de esquemas de memoria virtual almacena las ta-

blas de páginas también en la memoria virtual, en lugar de en la memoria real. Esto representa que las tablas de páginas están sujetas a paginación igual que cualquier otra página. Cuando un proceso está en ejecución, al menos parte de su tabla de páginas debe encontrarse en memoria, incluyendo la entrada de tabla de páginas de la página actualmente en ejecución. Algunos procesadores utilizan un esquema de dos niveles para organizar las tablas de páginas de gran tamaño. En este esquema, existe un directorio de páginas, en el cual cada entrada apuntaba a una tabla de páginas. De esta forma, si la extensión del directorio de páginas es X, y si la longitud máxima de una tabla de páginas es Y, entonces un proceso consistirá en hasta $X \geq Y$ páginas. Normalmente, la longitud máxima de la tabla de páginas se restringe para que sea igual a una página. Por ejemplo, el procesador Pentium utiliza esta estrategia.

La Figura 8.4 muestra un ejemplo de un esquema típico de dos niveles que usa 32 bits para la dirección. Asumimos un direccionamiento a nivel de byte y páginas de 4 Kbytes (2^{12}), por tanto el espacio de direcciones virtuales de 4 Gbytes (2^{32}) se compone de 2^{20} páginas. Si cada una de estas páginas se referencia por medio de una entrada la tabla de páginas (ETP) de 4-bytes, podemos crear una tabla de página de usuario con 2^{20} la ETP que requiere 4 Mbytes (2^{22} bytes). Esta enorme tabla de páginas de usuario, que ocupa 2^{10} páginas, puede mantenerse en memoria virtual y hacerse referencia desde una tabla de páginas raíz con 2^{10} PTE que ocuparía 4 Kbytes (2^{12}) de memoria principal. La Figura 8. 5 muestra los pasos relacionados con la traducción de direcciones para este esquema. La página raíz siempre se mantiene en la memoria principal. Los primeros 10 bits de la dirección virtual se pueden usar para indexar en la tabla de páginas raíz para encontrar la ETP para la página en la que está la tabla de páginas de usuario. Si la página no está en la memoria principal, se produce un fallo de página. Si la página está en la memoria principal, los siguientes 10 bits de la dirección virtual se usan para indexar la tabla de páginas de usuario para encontrar la ETP de la página a la cual se hace referencia desde la dirección virtual original.

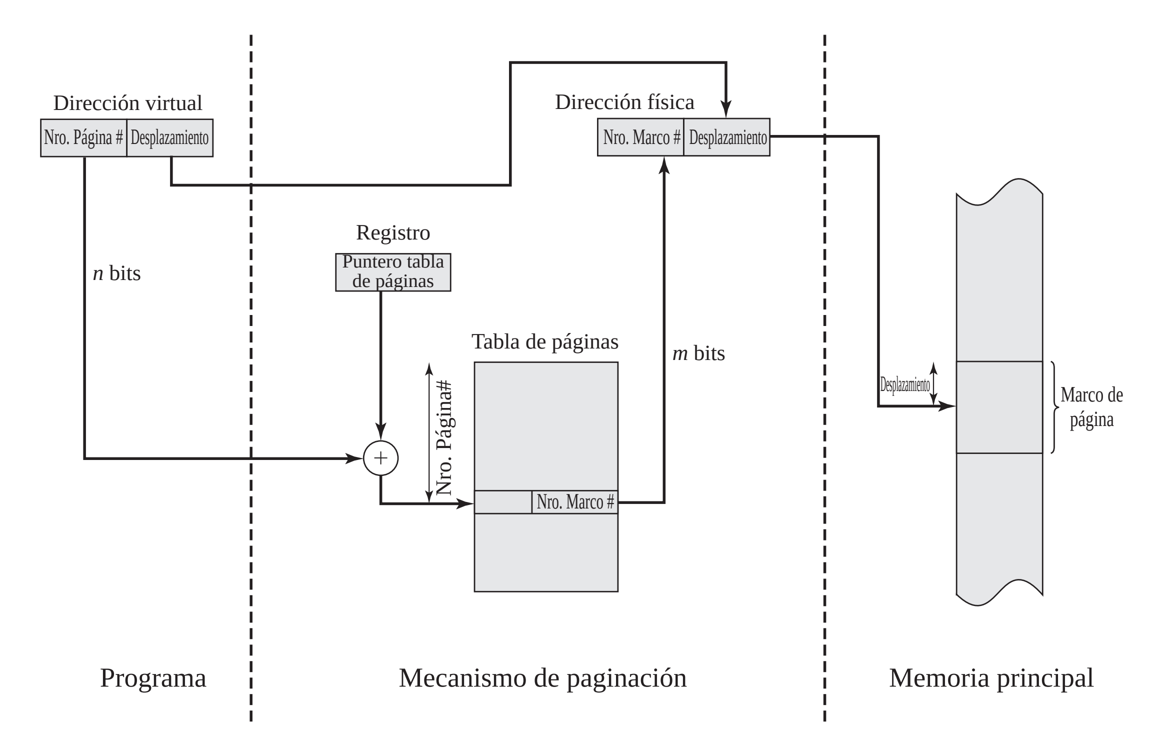


Figura 8.3. Traducción de direcciones en un sistema con paginación.

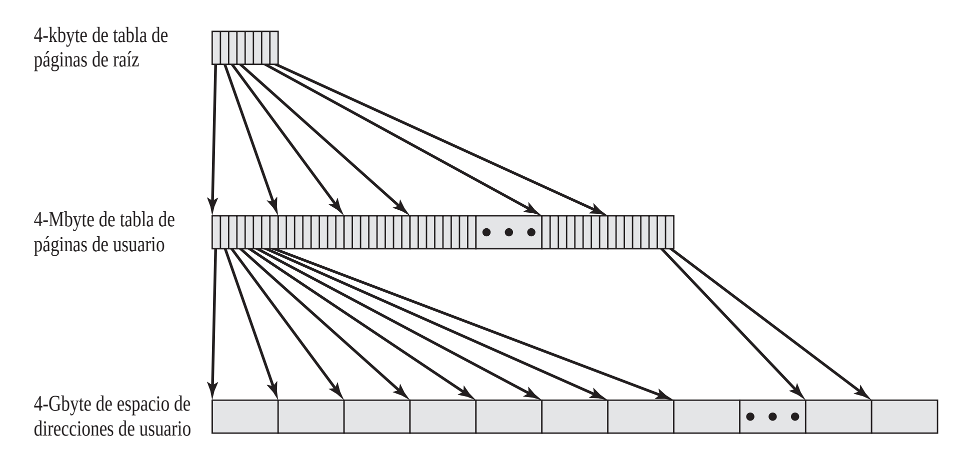


Figura 8.4. Una tabla de páginas jerárquica de dos niveles.

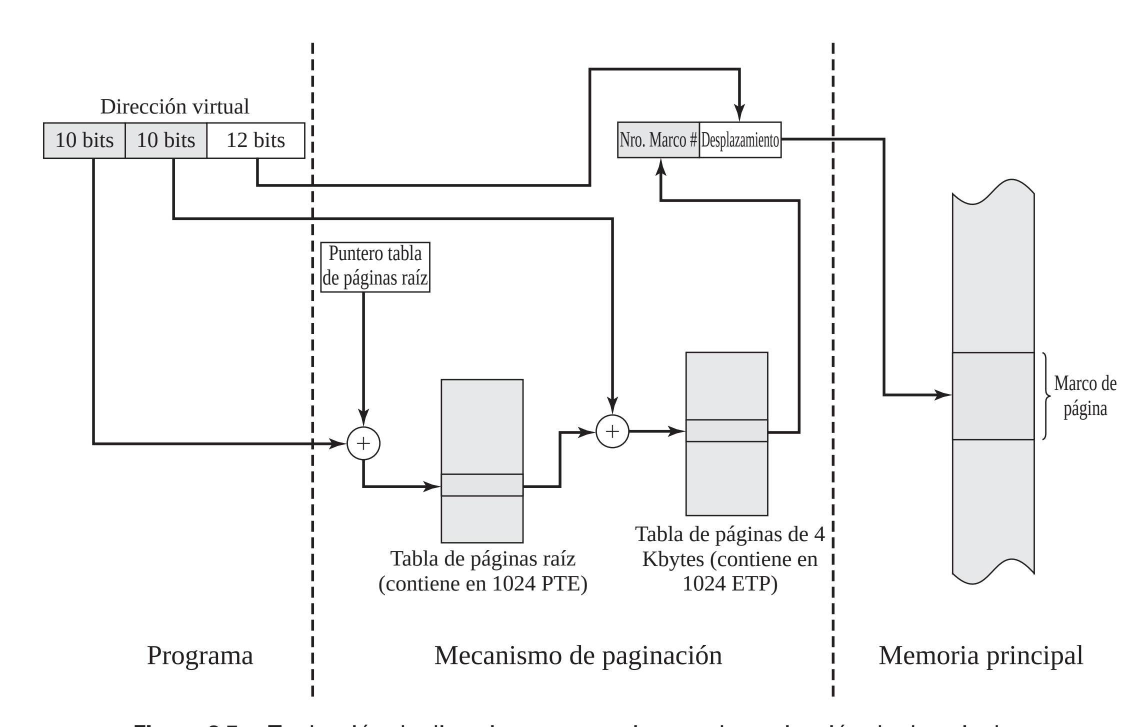


Figura 8.5. Traducción de direcciones en un sistema de paginación de dos niveles.

Tabla de páginas invertida. Una desventaja del tipo de tablas de páginas que hemos visto es que su tamaño es proporcional al espacio de direcciones virtuales.

Una estrategia alternativa al uso de tablas de páginas de uno o varios niveles es el uso de la estructura de **tabla de páginas invertida**. Variaciones de esta estrategia se han usado en arquitecturas como PowerPC, UltraSPARC, e IA-64. La implementación del sistema operativo Mach sobre RT-PC también la usa.

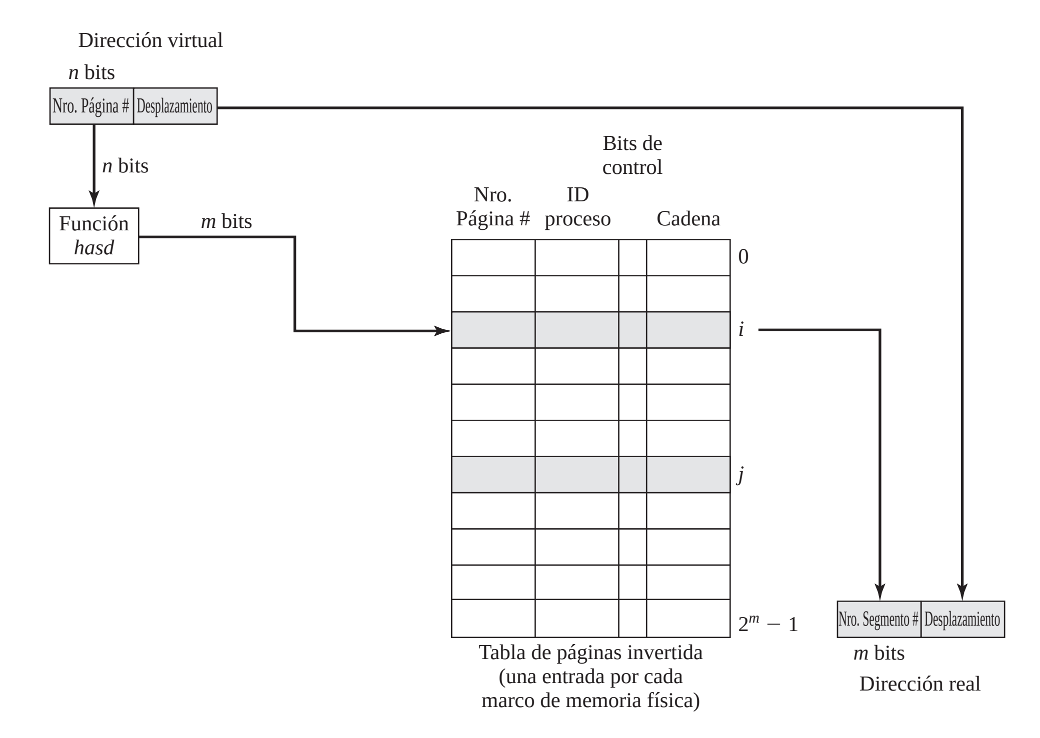


Figura 8.6. Estructura de tabla de páginas invertida.

En esta estrategia, la parte correspondiente al número de página de la dirección virtual se referencia por medio de un valor *hash* usando una función *hash* sencilla[1]. El valor *hash* es un puntero para la tabla de páginas invertida, que contiene las entradas de tablas de página. Hay una entrada en la tabla de páginas invertida por cada marco de página real en lugar de uno por cada página virtual. De esta forma, lo único que se requiere para estas tablas de página siempre es una proporción fija de la memoria real, independientemente del número de procesos o de las páginas virtuales soportadas. Debido a que más de una dirección virtual puede traducirse en la misma entrada de la tabla *hash*, una técnica de encadenamiento se utiliza para gestionar el desbordamiento. Las técnicas de *hashing* proporcionan habitualmente cadenas que no son excesivamente largas —entre una y dos entradas. La estructura de la tabla de páginas se denomina invertida debido a que se indexan sus entradas de la tabla de páginas por el número de marco en lugar de por el número de página virtual.

La Figura 8.6 muestra una implementación típica de la técnica de tabla de páginas invertida. Para un tamaño de memoria física de 2^m marcos, la tabla de páginas invertida contiene 2^m entradas, de forma que la entrada en la posición *i-esima* se refiere al marco *i*. La entrada en la tabla de páginas incluye la siguiente información:

- **Número de página.** Esta es la parte correspondiente al número de página de la dirección virtual.

---

[1] *Véase* Apéndice 8A para explicaciones sobre *hashing.*

- **Identificador del proceso.** El proceso que es propietario de está página. La combinación de número de página e identificador del proceso identifica a una página dentro del espacio de direcciones virtuales de un proceso en particular.
- **Bits de control.** Este campo incluye los *flags*, como por ejemplo, válido, referenciado, y modificado; e información de protección y cerrojos.
- **Puntero de la cadena.** Este campo es nulo (indicado posiblemente por un bit adicional) si no hay más entradas encadenadas en esta entrada. En otro caso, este campo contiene el valor del índice (número entre 0 y 2^{m-1}) de la siguiente entrada de la cadena.

En este ejemplo, la dirección virtual incluye un número de página de *n* bits, con $n > m$. La función *hash* traduce el número de página *n bits* en una cantidad de *m bits*, que se utiliza para indexar en la tabla de páginas invertida.

Buffer de traducción anticipada. En principio, toda referencia a la memoria virtual puede causar dos accesos a memoria física: uno para buscar la entrada la tabla de páginas apropiada y otro para buscar los datos solicitados. De esa forma, un esquema de memoria virtual básico causaría el efecto de duplicar el tiempo de acceso a la memoria. Para solventar este problema, la mayoría de esquemas de la memoria virtual utilizan una *cache* especial de alta velocidad para las entradas de la tabla de página, habitualmente denominada *buffer* de **traducción anticipada (*translation lookaside buffer* - TLB)**[2]. Esta *cache* funciona de forma similar a una memoria *cache* general (*véase* Capítulo 1) y contiene aquellas entradas de la tabla de páginas que han sido usadas de forma más reciente. La organización del hardware de paginación resultante se ilustra en la Figura 8.7. Dada una dirección virtual, el procesador primero examina la TLB, si la entrada de la tabla de páginas solicitada está presente (*acierto en TLB*), entonces se recupera el número de marco y se construye la dirección real. Si la entrada de la tabla de páginas solicitada no se encuentra (*fallo en la TLB*), el procesador utiliza el número de página para indexar la tabla de páginas del proceso y examinar la correspondiente entrada de la tabla de páginas. Si el bit de presente está puesto a 1, entonces la página se encuentra en memoria principal, y el procesador puede recuperar el número de marco desde la entrada de la tabla de páginas para construir la dirección real. El procesador también autorizará la TLB para incluir esta nueva entrada de tabla de páginas. Finalmente, si el bit presente no está puesto a 1, entonces la página solicitada no se encuentra en la memoria principal y se produce un fallo de acceso memoria, llamado **fallo de página**. En este punto, abandonamos el dominio del hardware para invocar al sistema operativo, el cual cargará la página necesaria y actualizada de la tabla de páginas.

La Figura 8.8 muestra un diagrama de flujo del uso de la TLB. Este diagrama de flujo muestra como si una página solicitada no se encuentra en la memoria principal, una interrupción de fallo de página hace que se invoque a la rutina de tratamiento de dicho fallo de página. Para mantener la simplicidad de este diagrama, no se ha mostrado el hecho de que el sistema operativo pueda activar otro proceso mientras la operación de E/S sobre disco se está realizando. Debido al principio de proximidad, la mayoría de referencias a la memoria virtual se encontrarán situadas en una página recientemente utiliza y por tanto, la mayoría de referencias invocarán una entrada de la tabla de páginas que se encuentra en la *cache*. Los estudios sobre la TLB de los sistemas VAX han demostrado que este esquema significa una importante mejora del rendimiento [CLAR85, SATY81].

Hay numerosos detalles adicionales relativos a la organización real de la TLB. Debido a que la TLB sólo contiene algunas de las entradas de toda la tabla de páginas, no es posible indexar simple-

---

[2] N. de T. Aunque la traducción más apropiada del *translation lookaside buffer* quizás sea la de *buffer* de traducción anticipada, en la literatura en castellano se utilizan las siglas TLB de forma generalizada para describir dicha memoria. Por ello, y para no causar confusión con otros textos, a lo largo del presente libro utilizaremos dichas siglas para referirnos a ella.

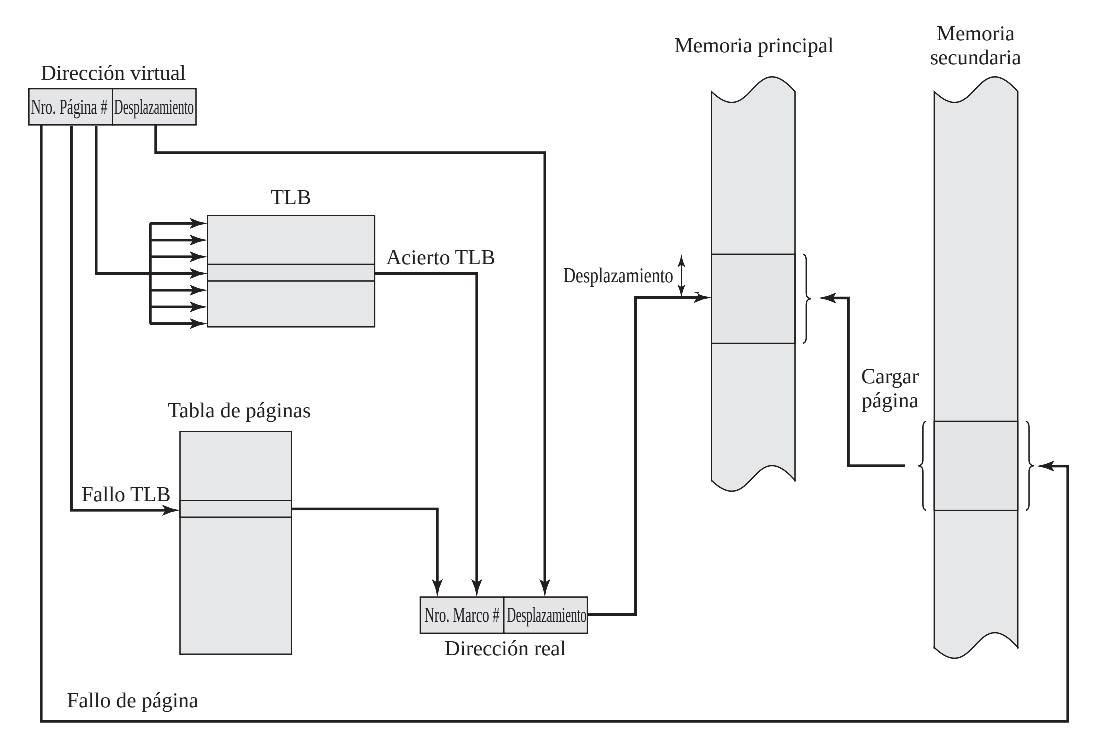


Figura 8.7. Uso de la TLB.

mente la TLB por medio de número página. En lugar de eso, cada entrada de la TLB debe incluir un número de página así como la entrada de la tabla de páginas completa. El procesador proporciona un hardware que permite consultar simultáneamente varias entradas para determinar si hay una conciencia sobre un número de página. Esta técnica se denomina **resolución asociativa (*asociative mapping*)** que contrasta con la resolución directa, o indexación, utilizada para buscar en la tabla de páginas en la Figura 8.9. El diseño de la TLB debe considerar también la forma mediante la cual las entradas se organizan en ella y qué entrada se debe reemplazar cuando se necesite traer una nueva entrada. Estos aspectos deben considerarse en el diseño de la cache hardware. Este punto no se contempla en este libro; el lector podrá consultar el funcionamiento del diseño de una cache para más detalle en, por ejemplo, [STAL03].

Para concluir, el mecanismo de memoria virtual debe interactuar con el sistema de *cache* (no la *cache* de TLB, sino la cache de la memoria principal). Esto se ilustra en la Figura 8.10. Una dirección virtual tendrá generalmente el formato número de página, desplazamiento. Primero, el sistema de memoria consulta la TLB para ver si se encuentra presente una entrada de tabla de página que coincide. Si es así, la dirección real (física) se genera combinando el número de marco con el desplazamiento. Si no, la entrada se busca en la tabla de páginas. Una vez se ha generado la dirección real, que mantiene el formato de etiqueta (*tag*)[3] y resto (*remainder*), se consulta la cache para ver si el bloque que contiene esa palabra se encuentra ahí. Si es así, se le devuelve a la CPU. Si no, la palabra se busca en la memoria principal.

---

[3] *Véase* en la Figura 1.17. Normalmente, una etiqueta son los bits situados más a la izquierda de una dirección real. Una vez más, para un estudio más detallado sobre las caches, se refiere al lector a [STAL03].

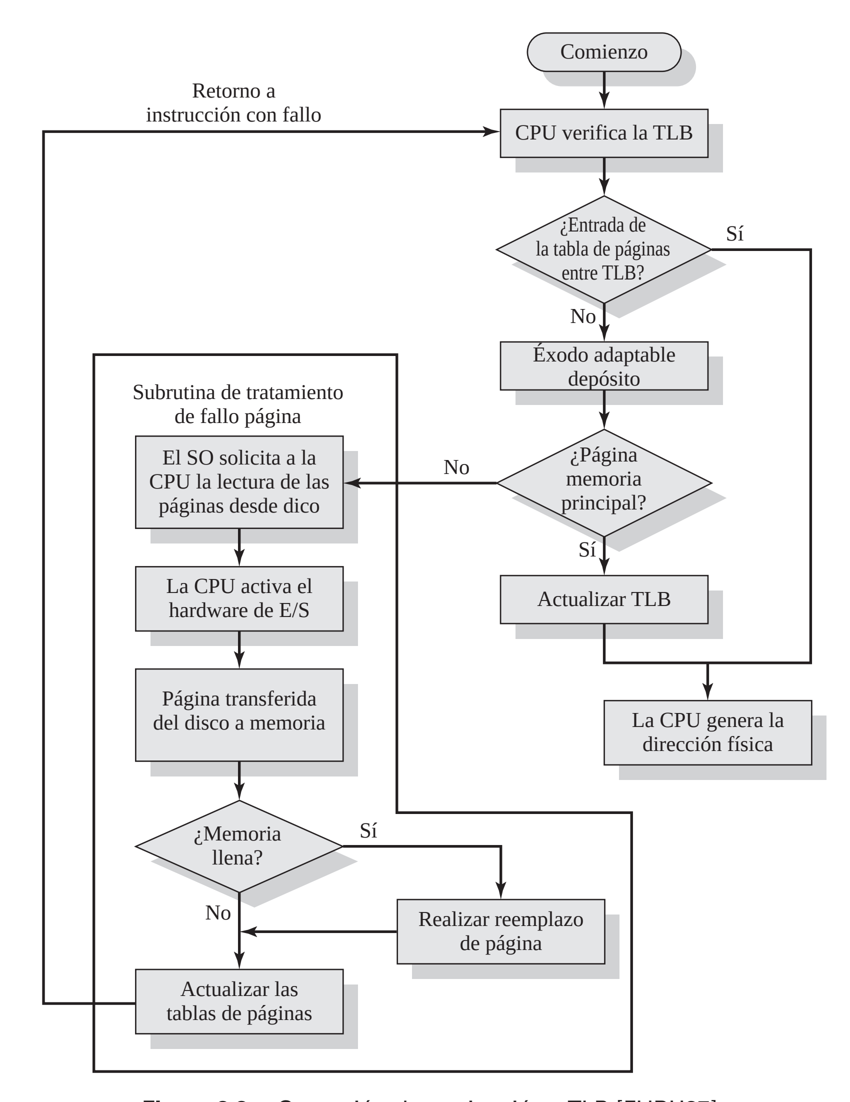


Figura 8.8. Operación de paginación y TLB [FURH87].

El lector podrá apreciar la complejidad del hardware de la CPU que participa en una referencia a memoria sencilla. La dirección virtual se traduce a una dirección real lo cual implica una referencia a la entrada de la tabla de páginas, que puede estar en la TLB, en la memoria principal, o en disco. La palabra referenciada puede estar en la cache, en la memoria principal, o en disco. Si dicha palabra referenciada se encuentra únicamente en disco, la página que contiene dicha palabra debe cargarse en la memoria principal y su bloque en la cache. Adicionalmente, la entrada en la tabla de páginas para dicha página debe actualizarse.

Tamaño de página. Una decisión de diseño hardware importante es el tamaño de página a usar. Hay varios factores a considerar. Por un lado, está la fragmentación interna. Evidentemente, cuanto mayor es el tamaño de la página, menor cantidad de fragmentación interna. Para optimizar el uso de la memoria principal, sería beneficioso reducir la fragmentación interna. Por otro lado, cuanto menor es la página, mayor número de páginas son necesarias para cada proceso. Un mayor número de páginas por proceso significa también mayores tablas de páginas. Para programas grandes en un entorno altamente multipro-

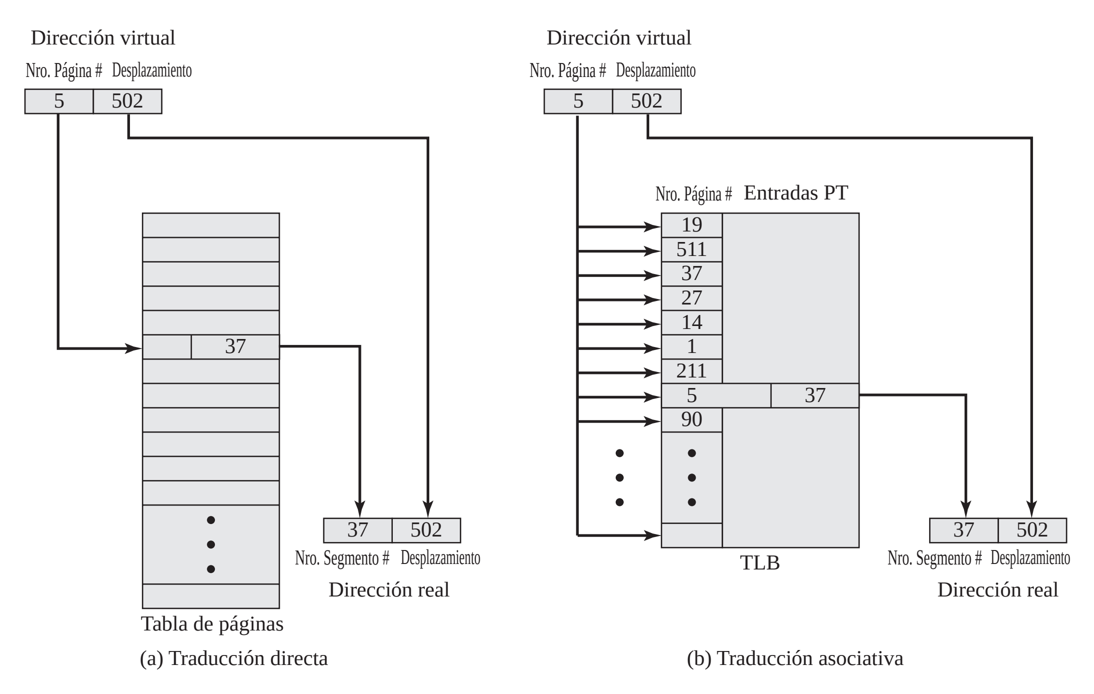


Figura 8.9. Resolución directa vs. asociativa para las entradas en la tabla de páginas.

gramado, esto significa que determinadas partes de las tablas de página de los procesos activos deben encontrarse en la memoria virtual, no en la memoria principal. Por tanto, puede haber un fallo de página doble para una referencia sencilla a memoria: el primero para atraer la tabla de página de la parte solicitada y el segundo para atraer la página del propio proceso. Otro factor importante son las características físicas de la mayoría de los dispositivos de la memoria secundaria, que son de tipo giratorio, favoreciendo tamaños de página grandes para mejorar la eficiencia de transferencia de bloques de datos.

Aumentando la complejidad de estos aspectos se encuentra el efecto que el tamaño de página tiene en relación a la posibilidad de que ocurra un fallo de página. Este comportamiento en términos generales, se encuentra recogido en la Figura 8.11a que se basa en el principio de proximidad. Si el tamaño de página es muy pequeño, de forma habitual habrá un número relativamente alto de páginas disponibles en la memoria principal para cada proceso. Después de un tiempo, las páginas en memoria contendrán las partes de los procesos a las que se ha hecho referencia de forma reciente. De esta forma, la tasa de fallos de página debería ser baja. A medida que el tamaño de páginas se incrementa, la página en particular contendrá información más lejos de la última referencia realizada. Así pues, el efecto del principio de proximidad se debilita y la tasa de fallos de página comienza a crecer. En algún momento, sin embargo, la tasa de fallos de página comenzará a caer a medida que el tamaño de la página se aproxima al tamaño del proceso completo (punto *P* en el diagrama). Cuando una única página contiene el proceso completo, no habrá fallos de página.

Una complicación adicional es que la tasa de fallos de página también viene determinada por el número de marcos asociados a cada proceso. La Figura 8.11b muestra que, para un tamaño de página fijo, la tasa de fallos cae a medida que el número de páginas mantenidas en la memoria principal crece[4]. Por

[4] El parámetro *W* representa el conjunto de trabajo, un concepto que se analizará en la Sección 8.2.

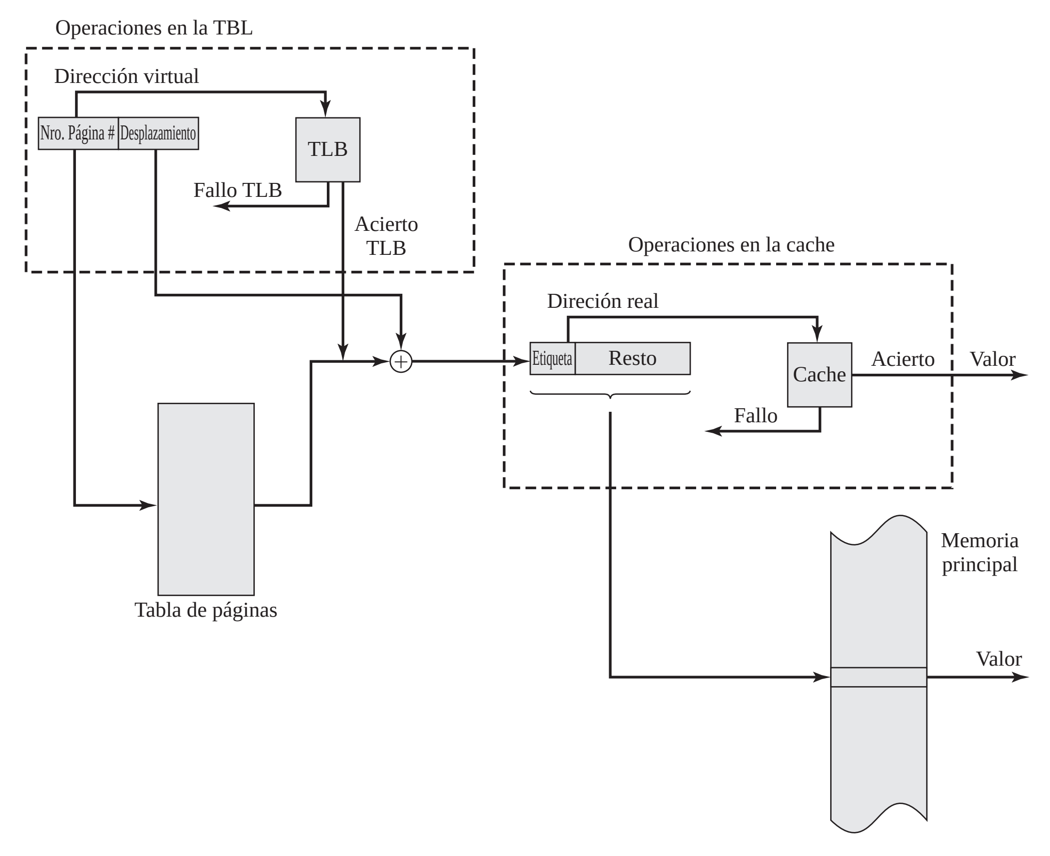


Figura 8.10. Operaciones en la TLB y en la *cache*.

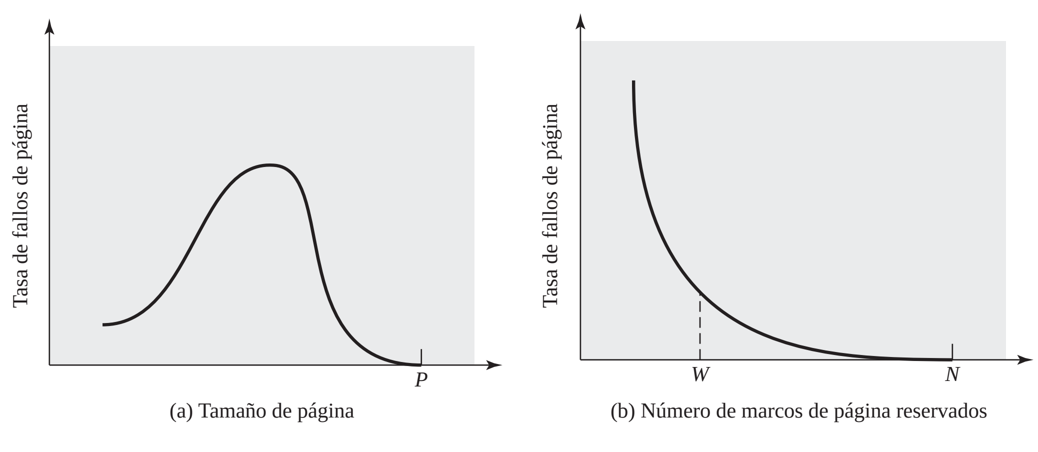


Figura 8.11. Comportamiento típico de la paginación de un programa.

tanto, una política software (la cantidad de memoria reservada por cada proceso) interactúa con decisiones de diseño del propio hardware (tamaño de página).

La Tabla 8 .2 contiene un listado de los tamaños de páginas que tienen determinadas arquitecturas.

Para concluir, el aspecto de diseño del tamaño página se encuentra relacionado con el tamaño de la memoria física y el tamaño del programa. Al mismo tiempo que la memoria principal está siendo cada vez más grande, el espacio de direcciones utilizado por las aplicaciones también crece. Esta tendencia resulta más evidente en ordenadores personales y estaciones de trabajo, donde las aplicaciones tienen una complejidad creciente. Por contra, diversas técnicas de programación actuales usadas para programas de gran tamaño tienden a reducir el efecto de la proximidad de referencias dentro un proceso [HUCK93]. Por ejemplo,

- Las técnicas de programación orientada a objetos motivan el uso de muchos módulos de datos y programas de pequeño tamaño con referencias repartidas sobre un número relativamente alto de objetos en un periodo de tiempo bastante corto.
- Las aplicaciones multihilo (*multithreaded*) pueden presentar cambios abruptos en el flujo de instrucciones y referencias a la memoria fraccionadas.

Tabla 8.2. Ejemplo de tamaños de página.

Computer	Tamaño de página
Atlas	512 palabras de 48-bits
Honeywell-Multics	1024 palabras de 36-bits
IBM 370/XA y 370/ESA	4 Kbytes
Familia VAX	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes hasta 16 Mbytes
UltraSPARC	8 Kbytes hasta 4 Mbytes
Pentium	4 Kbytes o 4 Mbytes
PowerPc	4 Kbytes
Itanium	4 Kbytes hasta 256 Mbytes

Para un tamaño determinado de una TLB, a medida que el tamaño del proceso crece y la proximidad de referencias decrece, el índice de aciertos en TLB se va reduciendo. Bajo estas circunstancias, la TLB se puede convertir en el cuello de botella del rendimiento (por ejemplo, *véase* [CHEN92]).

Una forma de incrementar el rendimiento en la TLB es utilizar una TLB de gran tamaño, con más entradas. Sin embargo, el tamaño de TLB interactúa con otros aspectos del diseño hardware, por ejemplo la *cache* de memoria principal o el número de accesos a memoria por ciclo de instrucción [TALL92]. Una de las principales pegas es que el tamaño de la TLB no tiene la misma tendencia de crecimiento que el tamaño de la memoria principal, en velocidad de crecimiento. Como alternativa se encuentra el uso de tamaños de página mayores de forma que cada entrada en la tabla de páginas referenciada en la TLB apunte a un bloque de memoria relativamente grande. Pero acabamos de ver que el uso de tamaños de página muy grandes puede significar la degradación del rendimiento.

Sobre estas consideraciones, un gran número de diseñadores han investigado la posibilidad de utilizar múltiples tamaños de página [TALL92, KHAL93], y diferentes arquitecturas de microprocesadores dan soporte a diversos tamaños de página, incluyendo MIPS R4000, Alpha, UltraSPARC, Pentium, e IA-64. Los tamaños de página múltiples proporcionan la flexibilidad necesaria para el uso de la TLB de forma eficiente. Por ejemplo, regiones contiguas de memoria de gran tamaño dentro del espacio direcciones del proceso, como las instrucciones del programa, se pueden proyectar sobre un reducido número de páginas de gran tamaño en lugar de un gran número de páginas de tamaño más pequeño, mientras que las pilas de los diferentes hilos se pueden alojar utilizando tamaños de página relativamente pequeños. Sin embargo, la mayoría de sistemas operativos comerciales aún soportan únicamente un tamaño de página, independientemente de las capacidades del hardware sobre el que están ejecutando. El motivo de esto se debe a que el tamaño de página afecta a diferentes aspectos del sistema operativo; por tanto, un cambio a un modelo de diferentes tamaños de páginas representa una tarea significativamente compleja (*véase* [GANA98] para más detalle).

SEGMENTACIÓN

Las implicaciones en la memoria virtual. La segmentación permite al programador ver la memoria como si se tratase de diferentes espacios de direcciones o segmentos. Los segmentos pueden ser de tamaños diferentes, en realidad de tamaño dinámico. Una referencia a la memoria consiste en un formato de dirección del tipo (número de segmento, desplazamiento).

Esta organización tiene un gran número de ventajas para el programador sobre los espacios de direcciones no segmentados:

1. Simplifica el tratamiento de estructuras de datos que pueden crecer. Si el programador no conoce a priori el tamaño que una estructura de datos en particular puede alcanzar es necesario hacer una estimación salvo que se utilicen tamaños de segmento dinámicos. Con la memoria virtual segmentada, a una estructura de datos se le puede asignar su propio segmento, y el sistema operativo expandirá o reducirá el segmento bajo demanda. Si un segmento que necesita expandirse se encuentre en la memoria principal y no hay suficiente tamaño, el sistema operativo poder mover el segmento a un área de la memoria principal mayor, si se encuentra disponible, o enviarlo a *swap*. En este último caso el segmento al que se ha incrementado el tamaño volverá a la memoria principal en la siguiente oportunidad que tenga.
2. Permite programas que se modifican o recopilan de forma independiente, sin requerir que el conjunto completo de programas se re-enlacen y se vuelvan a cargar. De nuevo, esta posibilidad se puede articular por medio de la utilización de múltiples segmentos.
3. Da soporte a la compartición entre procesos. El programador puede situar un programa de utilidad o una tabla de datos que resulte útil en un segmento al que pueda hacerse referencia desde otros procesos.
4. Soporta los mecanismos de protección. Esto es debido a que un segmento puede definirse para contener un conjunto de programas o datos bien descritos, el programador o el administrador de sistemas puede asignar privilegios de acceso de una forma apropiada.

Organización. En la exposición de la segmentación sencilla, indicamos que cada proceso tiene su propia tabla de segmentos, y que cuando todos estos segmentos se han cargado en la memoria principal, la tabla de segmentos del proceso se crea y se carga también en la memoria principal. Cada entrada de la tabla de segmentos contiene la dirección de comienzo del correspondiente segmento en la

memoria principal, así como la longitud del mismo. El mismo mecanismo, una tabla segmentos, se necesita cuando se están tratando esquemas de memoria virtual basados en segmentación. De nuevo, lo habitual es que haya una única tabla de segmentos por cada uno de los procesos. En este caso sin embargo, las entradas en la tabla de segmentos son un poco más complejas (Figura 8.2b). Debido a que sólo algunos de los segmentos del proceso pueden encontrarse en la memoria principal, se necesita un bit en cada entrada de la tabla de segmentos para indicar si el correspondiente segmento se encuentra presente en la memoria principal o no. Si indica que el segmento está en memoria, la entrada también debe incluir la dirección de comienzo y la longitud del mismo.

Otro bit de control en la entrada de la tabla de segmentos es el bit de modificado, que indica si los contenidos del segmento correspondiente se han modificado desde que se cargó por última vez en la memoria principal. Si no hay ningún cambio, no es necesario escribir el segmento cuando se reemplace de la memoria principal. También pueden darse otros bits de control. Por ejemplo, si la gestión de protección y compartición se gestiona a nivel de segmento, se necesitarán los bits correspondientes a estos fines.

El mecanismo básico para la lectura de una palabra de memoria implica la traducción de una dirección virtual, o lógica, consistente en un número de segmento y un desplazamiento, en una dirección física, usando la tabla de segmentos. Debido a que la tabla de segmentos es de tamaño variable, dependiendo del tamaño del proceso, no se puede suponer que se encuentra almacenada en un registro. En su lugar, debe encontrarse en la memoria principal para poder accederse. La Figura 8.12 sugiere una implementacione hardware de este esquema (nótese la similitud con la Figura 8.3). Cuando un proceso en particular está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos para dicho proceso. El número de segmento de la dirección virtual se utiliza para indexar esta tabla y para buscar la dirección de la memoria principal donde comienza dicho segmento. Ésta es añadida a la parte de desplazamiento de la dirección virtual para producir la dirección real solicitada.

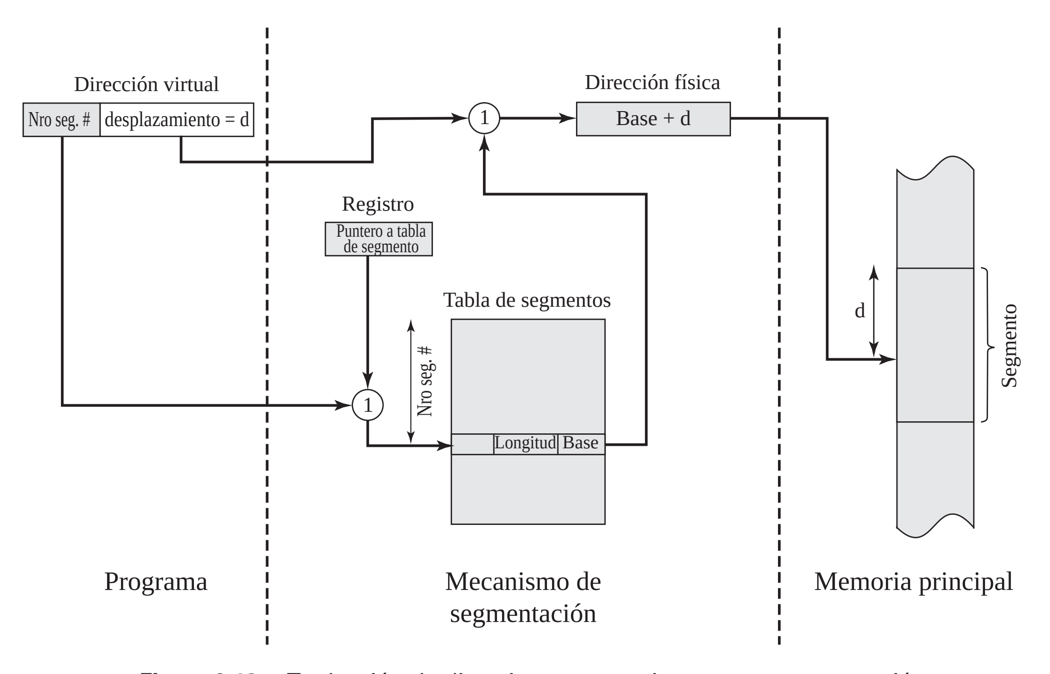


Figura 8.12. Traducción de direcciones en un sistema con segmentación.

PAGINACIÓN Y SEGMENTACIÓN COMBINADAS

Paginación y segmentación, cada una tiene sus propias ventajas. La paginación es transparente al programador y elimina la fragmentación externa, y por tanto proporciona un uso eficiente de la memoria principal. Adicionalmente, debido a que los fragmentos que se mueven entre la memoria y el disco son de un tamaño igual y prefijado, es posible desarrollar algoritmos de gestión de la memoria más sofisticados que exploten el comportamiento de los programas, como veremos más adelante. La segmentación sí es visible al programador y tiene los beneficios que hemos visto anteriormente, incluyendo la posibilidad de manejar estructuras de datos que crecen, modularidad, y dar soporte a la compartición y a la protección. Para combinar las ventajas de ambos, algunos sistemas por medio del hardware del procesador y del soporte del sistema operativo son capaces de proporcionar ambos.

En un sistema combinado de paginación/segmentación, el espacio de direcciones del usuario se divide en un número de segmentos, a discreción del programador. Cada segmento es, por su parte, dividido en un número de páginas de tamaño fijo, que son del tamaño de los marcos de la memoria principal. Si un segmento tiene longitud inferior a una página, el segmento ocupará únicamente una página. Desde el punto de vista del programador, una dirección lógica sigue conteniendo un número de segmento y un desplazamiento dentro de dicho segmento. Desde el punto de vista del sistema, el desplazamiento dentro del segmento es visto como un número de página y un desplazamiento dentro de la página incluida en el segmento.

La Figura 8.13 sugiere la estructura para proporcionar soporte o la combinación de paginación y segmentación (nótese la similitud con la Figura 8.5). Asociada a cada proceso existe una tabla de segmentos y varias tablas de páginas, una por cada uno de los segmentos. Cuando un proceso está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos de dicho proceso. A partir de la dirección virtual, el procesador utiliza la parte correspondiente al número de segmento para indexar dentro de la tabla de segmentos del proceso para encontrar la tabla de pági-

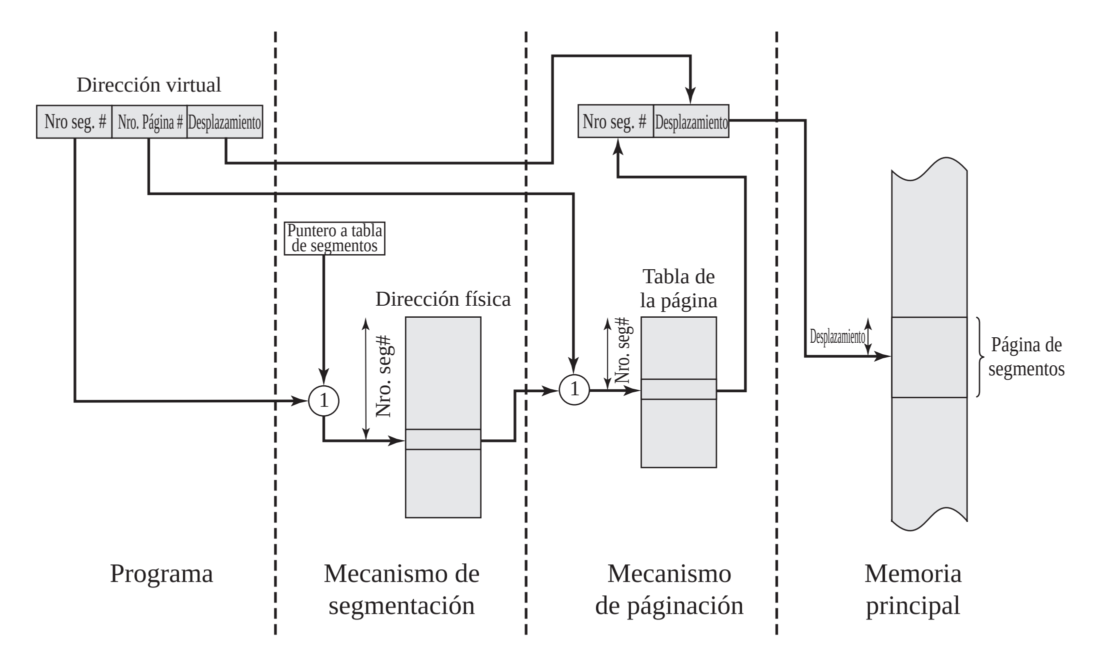


Figura 8.13. Traducción de direcciones en un sistema con segmentación/paginación.

nas de dicho segmento. Después, la parte correspondiente al número de página de la dirección virtual original se utiliza para indexar la tabla de páginas y buscar el correspondiente número de marco. Éste se combina con el desplazamiento correspondiente de la dirección virtual para generar la dirección real requerida.

En la Figura 8.2c se muestran los formatos de la entrada en la tabla de segmentos y de la entrada en la tabla de páginas. Como antes, la entrada en la tabla de segmentos contiene la longitud del segmento. También contiene el campo base, que ahora hace referencia a la tabla de páginas. Los bits de presente y modificado no se necesitan debido a que estos aspectos se gestionan a nivel de página. Otros bits de control sí pueden utilizarse, a efectos de compartición y protección. La entrada en la tabla de páginas es esencialmente la misma que para el sistema de paginación puro. El número de página se proyecta en su número de marco correspondiente si la página se encuentra presente en la memoria. El bit de modificado indica si la página necesita escribirse cuando se expulse del marco de página actual. Puede haber otros bits de control relacionados con la protección u otros aspectos de la gestión de la memoria.

PROTECCIÓN Y COMPARTICIÓN

La segmentación proporciona una vía para la implementación de las políticas de protección y compartición. Debido a que cada entrada en la tabla de segmentos incluye la longitud así como la dirección base, un programa no puede, de forma descontrolada, acceder a una posición de memoria principal más allá de los límites del segmento. Para conseguir compartición, es posible que un segmento se encuentre referenciado desde las tablas de segmentos de más de un proceso. Los mecanismos están, por supuesto, disponibles en los sistemas de paginación. Sin embargo, en este caso la estructura de páginas de un programa y los datos no son visible para el programador, haciendo que la especificación de la protección y los requisitos de compartición sean menos cómodos. La Figura 8.14 ilustra los tipos de relaciones de protección que se pueden definir en dicho sistema.

También es posible proporcionar mecanismos más sofisticados. Un esquema habitual es utilizar la estructura de protección en anillo, del tipo que indicamos en el Capítulo 3 (Problema 3.7). En este esquema, los anillos con números bajos, o interiores, disfrutan de mayores privilegios que los anillos con numeraciones más altas, o exteriores. Normalmente, el anillo 0 se reserva para funciones del núcleo del sistema operativo, con las aplicaciones en niveles superiores. Algunas utilidades o servicios de sistema operativo pueden ocupar un anillo intermedio. Los principios básicos de los sistemas en anillo son los siguientes:

- Un programa pueda acceder sólo a los datos residentes en el mismo anillo o en anillos con menos privilegios.
- Un programa puede invocar servicios residentes en el mismo anillo o anillos con más privilegios.

8.2. SOFTWARE DEL SISTEMA OPERATIVO

El diseño de la parte de la gestión de la memoria del sistema operativo depende de tres opciones fundamentales a elegir:

- Si el sistema usa o no técnicas de memoria virtual.
- El uso de paginación o segmentación o ambas.
- Los algoritmos utilizados para los diferentes aspectos de la gestión de la memoria.

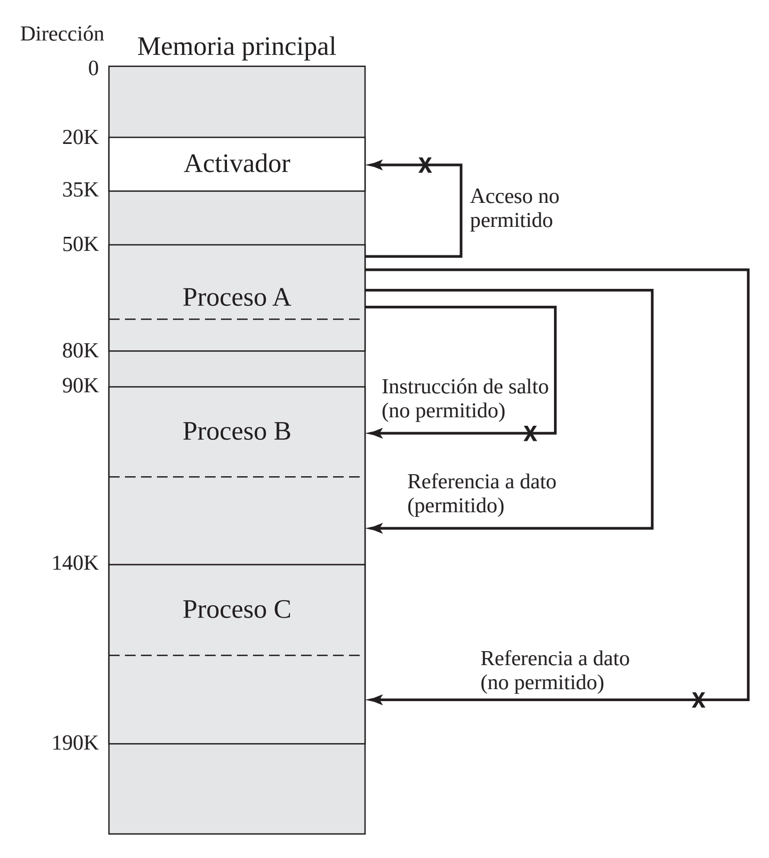


Figura 8.14. Relaciones de protección entre segmentos.

Las elecciones posibles para las dos primeras opciones dependen de la plataforma hardware disponible. Así, las primeras implantaciones de UNIX no proporcionaban memoria virtual porque los procesadores sobre los cuales ejecutaban no daban soporte para paginación o segmentación. Ninguna de estas técnicas es abordable sin una plataforma hardware para traducción de direcciones y otras funciones básicas.

Hay también dos comentarios adicionales sobre estas dos primeras opciones: primero, con la excepción de los sistemas operativos de algunas plataformas como los ordenadores personales antiguos, como MS-DOS, y de otros sistemas de carácter especializado, todos los sistemas operativos importantes proporcionan memoria virtual. Segundo, los sistemas de segmentación pura son en la actualidad realmente escasos. Cuando la segmentación se combina con paginación, la mayoría de los aspectos de la gestión de la memoria relativos al diseño sistema operativo se encuadran en el área de la paginación[5]. De esta forma, en esta sección nos concentraremos en los aspectos asociados a la paginación.

Las elecciones relativas a la tercera opción entran dentro del dominio del software del sistema operativo y son el objeto de esta sección. La Tabla 8.3 muestra los elementos de diseño clave que se

---

[5] La protección y la compartición, en un sistema combinado de segmentación/paginación, se suelen delegar habitualmente a nivel de segmento. Abordaremos estas cuestiones en capítulos posteriores.

Tabla 8.3. Políticas del sistema operativo para la memoria virtual.

Política de recuperación	**Gestión del conjunto residente**
Bajo demanda	Tamaño del conjunto residente
Paginación adelantada	Fijo
	Variable
Política de ubicación	Ámbito de reemplazo
	Global
Política de reemplazo	Local
Algoritmos básicos	
Óptimo	**Política de limpieza**
FIFO	Bajo demanda
Usada menos recientemente (LRU)	Limpieza adelantada
Del reloj	**Control de carga**
Buffers de página	Grado de multiprogramación

van a examinar. En cada caso, el aspecto central es el rendimiento: Se tratará de minimizar la tasa de ocurrencia de fallos de página, porque los fallos de página causan una considerable sobrecarga sobre el software. Como mínimo, esta sobrecarga incluye la decisión de qué página o páginas residentes se van a reemplazar, y la E/S del intercambio o *swap* de dichas páginas. También, el sistema operativo debe planificar la ejecución de otro proceso durante la operación de E/S de la página, causando un cambio de contexto. De la misma forma, intentaremos organizar determinados aspectos de forma que, durante el tiempo de ejecución de un proceso, la probabilidad de hacer referencia a una palabra en una página que no se encuentre presente sea mínima. En todas estas áreas indicadas en la Tabla 8.3, no existe una política que sea mejor que todas las demás. Como se verá, la tarea de gestión de la memoria en un entorno de paginación es endiabladamente compleja. Adicionalmente, el rendimiento de un conjunto de políticas en particular depende del tamaño de la memoria, de la velocidad relativa de la memoria principal y secundaria, del tamaño y del número de procesos que están compitiendo por los recursos, y del comportamiento en ejecución de los diferentes programas de forma individual. Esta última característica depende de la naturaleza de la aplicación, el lenguaje de programación y el compilador utilizado, el estilo del programador que lo escribió, y, para un programa interactivo, el comportamiento dinámico del usuario. Así pues, el lector no debe esperar de ningún modo una respuesta definitiva aquí. Para sistemas pequeños, el diseño de sistema operativo debe intentar elegir un conjunto de políticas que parezcan funcionar «bien» sobre una amplia gama de condiciones, basándose en el conocimiento sobre el estado actual del sistema. Para grandes sistemas, particularmente *mainframes*, el sistema operativo debe incluir herramientas de monitorización y control que permitan al administrador de la instalación ajustar ésta para obtener «buenos» resultados en base a las condiciones de la instalación.

POLÍTICA DE RECUPERACIÓN

La política de recuperación determina cuándo una página se trae a la memoria principal. Las dos alternativas habituales son bajo demanda y paginación adelantada (*prepaging*). Con **paginación bajo demanda**, una página se trae a memoria sólo cuando se hace referencia a una posición en dicha página. Si el resto de elementos en la política de gestión de la memoria funcionan correctamente, ocurriría

lo siguiente. Cuando un proceso se arranca inicialmente, va a haber una ráfaga de fallos de página. Según se van trayendo más y más páginas a la memoria, el principio de proximidad sugiere que las futuras referencias se encontrarán en las páginas recientemente traídas. Así, después de un tiempo, la situación se estabilizará y el número de fallos de página caerá hasta un nivel muy bajo.

Con **paginación adelantada (*prepaging*)**, se traen a memoria también otras páginas, diferentes de la que ha causado el fallo de página. La paginación adelantada tiene en cuenta las características que tienen la mayoría de dispositivos de memoria secundaria, tales como los discos, que tienen tiempos de búsqueda y latencia de rotación. Si las páginas de un proceso se encuentran almacenadas en la memoria secundaria de forma contigua, es mucho más eficiente traer a la memoria un número de páginas contiguas de una vez, en lugar de traerlas una a una a lo largo de un periodo de tiempo más amplio. Por supuesto, esta política es ineficiente si la mayoría de las páginas que se han traído no se referencian a posteriori.

La política de paginación adelantada puede emplearse bien cuando el proceso se arranca, en cuyo caso el programador tiene que designar de alguna forma las páginas necesarias, o cada vez que ocurra un fallo de página. Este último caso es el más apropiado porque resulta completamente invisible al programador. Sin embargo, la completa utilidad de la paginación adelantada no se encuentra reconocida [MAEK87].

La paginación adelantada no se debe confundir con el *swapping*. Cuando un proceso se saca de la memoria y se le coloca en estado suspendido, todas sus páginas residentes se expulsan de la memoria. Cuando el proceso se recupera, todas las páginas que estaban previamente en la memoria principal retornan a ella.

POLÍTICA DE UBICACIÓN

La política de ubicación determina en qué parte de la memoria real van a residir las porciones de la memoria de un proceso. En los sistema de segmentación puros, la política de ubicación es un aspecto de diseño muy importante; políticas del estilo mejor ajuste, primer ajuste, y similares que se discutieron en el Capítulo 7, son las diferentes alternativas. Sin embargo, para sistemas que usan o bien paginación pura o paginación combinada con segmentación, la ubicación es habitualmente irrelevante debido a que el hardware de traducción de direcciones y el hardware de acceso a la memoria principal pueden realizar sus funciones en cualquier combinación de página-marco con la misma eficiencia.

Existe otro entorno en el cual la ubicación tiene implicación importante, y es un tema de investigación y desarrollo. En aquellos sistemas llamados multiprocesadores de acceso a la memoria no uniforme (*nonuniform memory access multiprocessors*-NUMA), la memoria distribuida compartida de la máquina puede referenciarse por cualquier otro procesador dentro de la misma máquina, pero con un tiempo de acceso dependiente de la localización física y que varía con la distancia entre el procesador y el módulo de la memoria. De esta forma, el rendimiento depende significativamente de la distancia a la cual reside el dato en relación al procesador que va a utilizar [LARO92, BOLO89, COX89]. Para sistemas NUMA, una estrategia de ubicación automática aceptable es aquella que asigna las páginas al módulo de la memoria que finalmente proporcionará mejor rendimiento.

POLÍTICA DE REEMPLAZO

En la mayoría de los libros sobre sistemas operativos, el tratamiento de la gestión de la memoria incluye una sección titulada «política de reemplazo», que trata de la selección de una página en la memoria principal como candidata para reemplazarse cuando se va traer una nueva página. Este tema es, a menudo, difícil de explicar debido a que hay varios conceptos interrelacionados:

- ¿Cuántos marcos de página se van a reservar para cada uno de los procesos activos?
- Si el conjunto de páginas que se van a considerar para realizar el reemplazo se limita a aquellas del mismo proceso que ha causado el fallo de página o, si por el contrario, se consideran todos los marcos de página de la memoria principal.
- Entre el conjunto de páginas a considerar, qué página en concreto es la que se va a reemplazar.

Nos referimos a los dos primeros conceptos como la *gestión del conjunto residente*, que se trata en la siguiente subsección, y se ha reservado el término *política de reemplazo* para el tercer concepto, que se discutirá en esta misma subsección.

El área de políticas de reemplazo es probablemente el aspecto de la gestión de la memoria que ha sido más estudiado. Cuando todos los marcos de la memoria principal están ocupados y es necesario traer una nueva página para resolver un fallo de página, la política de reemplazo determina qué página de las que actualmente están en memoria va a reemplazarse. Todas las políticas tienen como objetivo que la página que va a eliminarse sea aquella que tiene menos posibilidades de volver a tener una referencia en un futuro próximo. Debido al principio de proximidad de referencia, existe a menudo una alta correlación entre el histórico de referencias recientes y los patrones de referencia en un futuro próximo. Así, la mayoría de políticas tratan de predecir el comportamiento futuro en base al comportamiento pasado. En contraprestación, se debe considerar que cuanto más elaborada y sofisticada es una política de reemplazo, mayor va a ser la sobrecarga a nivel software y hardware para implementarla.

Bloqueo de marcos. Es necesario mencionar una restricción que se aplica a las políticas de reemplazo antes de indagar en los diferentes algoritmos: algunos marcos de la memoria principal pueden encontrarse bloqueados. Cuando un marco está bloqueado, la página actualmente almacenada en dicho marco no puede reemplazarse. Gran parte del núcleo del sistema operativo se almacena en marcos que están bloqueados, así como otras estructuras de control claves. Adicionalmente, los *buffers* de E/S y otras áreas de tipo critico también se ponen en marcos bloqueados en la memoria principal. El bloqueo se puede realizar asociando un bit de bloqueo a cada uno de los marcos. Este bit se puede almacenar en la tabla de marcos o también incluirse en la tabla de páginas actual.

Algoritmos básicos. Independientemente de la estrategia de gestión del conjunto residente (que se discutirá en la siguiente subsección), existen ciertos algoritmos básicos que se utilizan para la selección de la página a reemplazar. Los algoritmos de reemplazo que se han desarrollando a lo largo de la literatura son:

- Óptimo.
- Usado menos recientemente (*least recently used*-LRU).
- FIFO (*first-in-first-out*).
- Reloj.

La **política óptima** de selección tomará como reemplazo la página para la cuál el instante de la siguiente referencia se encuentra más lejos. Se puede ver que para esta política los resultados son el menor número de posibles fallos de página [BELA66]. Evidentemente, esta política es imposible de implementar, porque requiere que el sistema operativo tenga un perfecto conocimiento de los eventos futuros. Sin embargo se utiliza como un estándar apartir del cual contrastar algoritmos reales.

La Figura 8.15 proporciona un ejemplo de la política óptima. El ejemplo asume una reserva de marcos fija (tamaño del conjunto residente fijo) para este proceso de un total de tres marcos. La eje-

cución de proceso requiere la referencia de cinco páginas diferentes. El flujo de páginas referenciadas por el programa antes citado es el siguiente:

2 3 2 1 5 2 4 5 3 2 5 2

lo cual representa que la primera página a la que se va a hacer referencia es la 2, la segunda página la 3, y así en adelante. La política óptima produce tres fallos de página después de que la reserva de marcos se haya ocupado completamente.

La política de reemplazo de la página **usada menos recientemente (LRU)** seleccionará como candidata la página de memoria que no se haya referenciado desde hace más tiempo. Debido al principio de proximidad referenciada, esta página sería la que tiene menos probabilidad de volverá a tener referencias en un futuro próximo. Y, de hecho, la política LRU proporciona unos resultados casi tan buenos como la política óptima. El problema con esta alternativa es la dificultad en su implementación. Una opción sería etiquetar cada página con el instante de tiempo de su última referencia; esto podría ser en cada una de las referencias a la memoria, bien instrucciones o datos. Incluso en el caso de que el hardware diera soporte a dicho esquema, la sobrecarga sería tremenda. De forma alternativa se puede mantener una pila de referencias a páginas, que igualmente es una opción costosa.

La Figura 8.15 muestra un ejemplo del comportamiento de LRU, utilizando el mismo flujo de referencias a páginas que en el ejemplo de la política óptima. En este ejemplo, se producen cuatro fallos de página.

La **política FIFO** trata los marcos de página ocupados como si se tratase de un *buffer* circular, y las páginas se remplazan mediante una estrategia cíclica de tipo round-robin. Todo lo que se necesita

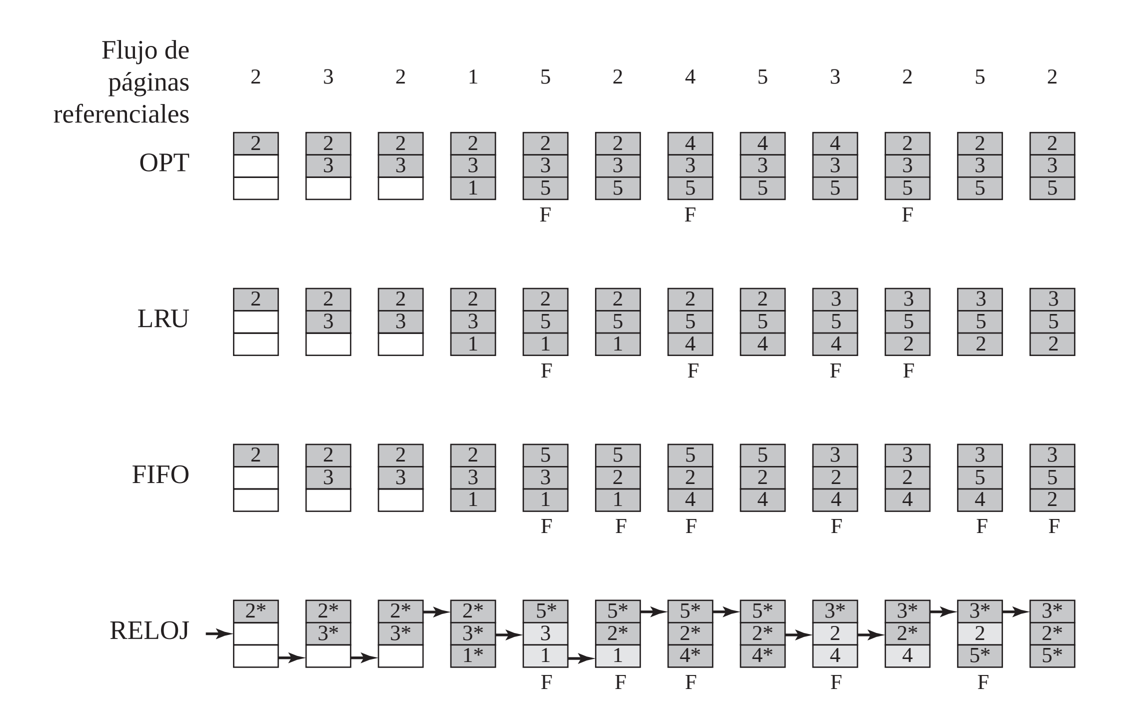


Figura 8.15. Comportamiento de cuatro algoritmos de reemplazo de páginas.

es un puntero que recorra de forma circular los marcos de página del proceso. Por tanto, se trata de una de las políticas de reemplazo más sencilla de implementar. El razonamiento tras este modelo, además de su simplicidad, es el reemplazo de la página que lleva en memoria más tiempo: una página traída a la memoria hace mucho tiempo puede haber dejado de utilizarse. Este razonamiento a menudo es erróneo, debido a que es habitual que en los programas haya una zona del mismo o regiones de datos que son utilizados de forma intensiva durante todo el tiempo de vida del proceso. Esas páginas son expulsadas de la memoria y traídas de nuevo de forma repetida por un algoritmo de tipo FIFO.

Continuando con el mismo ejemplo de la Figura 8.15, la política FIFO genera un total de seis fallos de página. Nótese que el algoritmo LRU reconoce que a las páginas 2 y 5 se hace referencia con mayor frecuencia que a cualquier otra página, mientras que FIFO no lo hace.

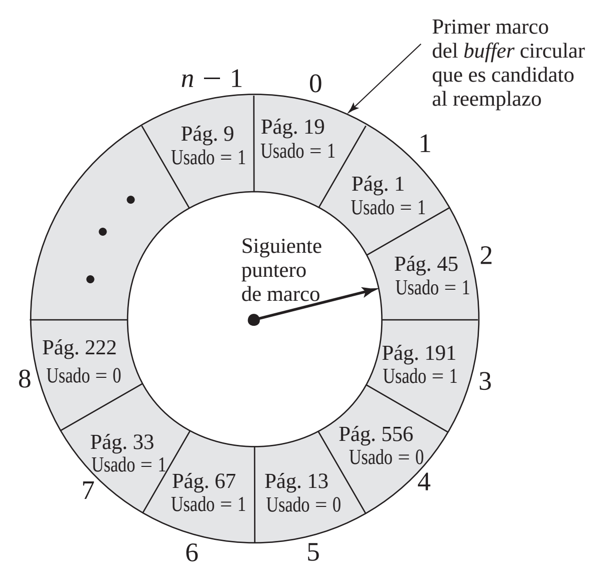
Mientras que la política LRU alcanza unos resultados similares a la política óptima, es difícil de implementar e impone una sobrecarga significativa. Por otro lado, la política FIFO es muy sencilla de implementar pero su rendimiento es relativamente pobre. A lo largo de los años, los diseñadores de sistemas operativos han intentado un gran número de algoritmos diferentes para aproximarse a los resultados obtenidos por LRU e intentando imponer una sobrecarga más reducida. Muchos de estos algoritmos son variantes del esquema denominado **política del reloj**.

En su forma más sencilla la política del reloj requiere la inclusión de un bit adicional en cada uno de los marcos de página, denominado bit de usado. Cuando una página se trae por primera vez a la memoria, el bit de usado de dicho marco se pone a 1. En cualquier momento que la página vuelva a utilizarse (después de la referencia generada con el fallo de página inicial) su bit de usado se pone a 1. Para el algoritmo de reemplazo de páginas, el conjunto de todas las páginas que son candidatas para reemplazo (de este proceso: ámbito local; toda la memoria principal: ámbito global[6]) se disponen como si se tratase de un *buffer* circular, al cual se asocia un puntero. Cuando se reemplaza una página, el puntero indica el siguiente marco del *buffer* justo después del marco que acaba de actualizarse. Cuando llega el momento de reemplazar una página, el sistema operativo recorre el *buffer* para encontrar un marco con su bit de usado a 0. Cada vez que encuentra un marco con el bit de usado a 1, se reinicia este bit a 0 y se continúa. Si alguno de los marcos del *buffer* tiene el bit de usado a 0 al comienzo de este proceso, el primero de estos marcos que se encuentre se seleccionará para reemplazo. Si todos los marcos tienen el bit a 1, el puntero va a completar un ciclo completo a lo largo del *buffer*, poniendo todo los bits de usado a 0, parándose en la posición original, reemplazando la página en dicho marco. *Véase* que esta política es similar a FIFO, excepto que, en la política del reloj, el algoritmo saltará todo marco con el bit de usado a 1. La política se domina política del reloj debido a que se pueden visualizar los marcos de página como si estuviesen distribuidos a lo largo del círculo. Un gran número de sistemas operativos han empleado alguna variante de esta política sencilla del reloj (por ejemplo, Multics [CORB68]).

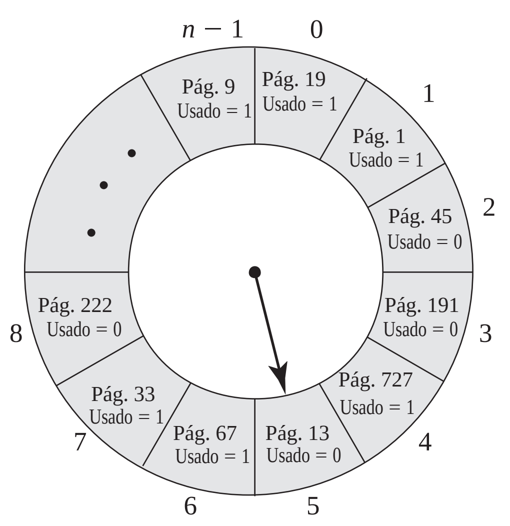
La Figura 8.16 plantea un ejemplo del mecanismo de la política del reloj. Un *buffer* circular con *n* marcos de memoria principal que se encuentran disponibles para reemplazo de la página. Antes del comienzo del reemplazo de una página del *buffer* por la página entrante 727, el puntero al siguiente marco apunta al marco número 2, que contiene la página 45. En este momento la política del reloj comienza a ejecutarse. Debido a que el bit usado de la página 45 del marco 2 es igual a 1, esta página no se reemplaza. En vez de eso, el bit de usado se pone a 0 y el puntero avanza. De forma similar la página 191 en el marco 3 tampoco se remplazará; y su bit de usado se pondrá a 0, avanzando de nuevo el puntero. En el siguiente marco, el marco número 4, el bit de usado está a 0. Por tanto, la página 556 se reemplazará por la página 727. El bit de usado se pone a 1 para este marco y el puntero avanza hasta el marco 5, completando el procedimiento de reemplazo de página.

---

[6] El concepto de ámbito se discute en la subsección «Ámbito de reemplazo», más adelante.



(a) Estado del *buffer* justo ante del reemplazo de una página



(b) Estado del *buffer* justo después del siguiente reemplazo de página

Figura 8.16. Ejemplo de operación de la política de reemplazo del reloj.

El comportamiento de la política del reloj se encuentra ilustrado en la Figura 8.15. La presencia de un asterisco indica que el correspondiente bit de usado es igual a 1, y la flecha indica cuál es la posición actual del puntero. Nótese que la política del reloj intenta proteger los marcos 2 y 5 de un posible reemplazo.

La Figura 8.17 muestra los resultados del experimento realizado por [BAER80], que compara los cuatro algoritmos que se han comentado; se asume que el número de marcos de página asignados a cada proceso es fijo. El resultado se basa en la ejecución de $0,25 \times 10^6$ referencias en un programa FORTRAN, utilizando un tamaño de página de 256 palabras. Baer ejecutó el experimento con unas reservas de 6, 8, 10, 12, y 14 marcos. Las diferencias entre las cuatro políticas son más palpables

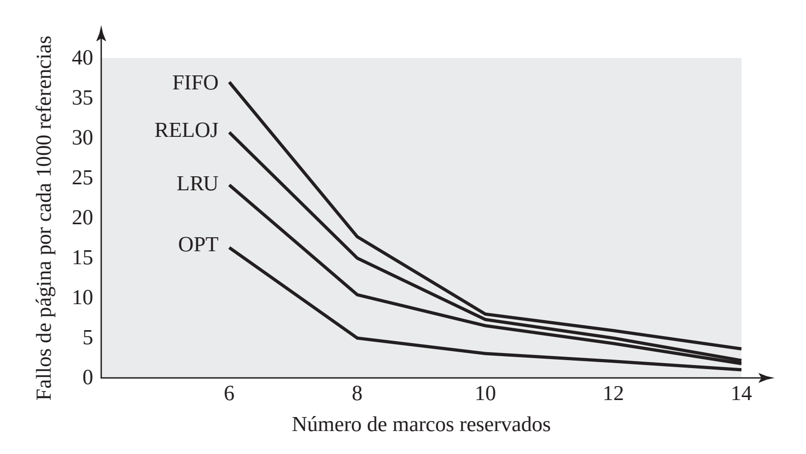


Figura 8.17. Comparativa de algoritmos de reemplazo local con reserva de marcos fija.

cuando el número de marcos reservados es pequeño, estando FIFO por encima en un factor de 2 veces peor que el óptimo. Las cuatro curvas mantienen la misma forma de comportamiento que el ideal mostrado en la Figura 8.11b. Con intención de ejecutar de forma eficiente, sería deseable encontrarse en el lado derecho de la curva (con una tasa de fallos de página pequeña) mientras que al mismo tiempo se mantiene una necesidad de reservar relativamente pocos marcos (hacia el lado izquierdo de la curva). Estas dos restricciones indican que el modo deseable de operación estaría aproximadamente en la mitad de la curva.

[FINK88] tanbién reporta unos resultados prácticamente idénticos, de nuevo mostrando una desviación máxima en torno a un factor de 2. La estrategia de Finkel consistía en simular el efecto de varias políticas en una cadena de referencias a páginas generada sintéticamente de un total de 10.000 referencias seleccionadas dentro del espacio virtual de 100 páginas. Para aproximarse a los efectos del principio de próximidad de referencia, se impuso el uso de una distribución exponencial de probabilidad para hacer referencia a una página en concreto. Finkel indica que se podría concluir que no tiene mucho sentido elaborar algoritmos de reemplazo de páginas cuando sólo hay un factor de 2 en juego. Pero remarca que esta diferencia puede tener un efecto considerable en los requisitos de memoria principal (si se quiere evitar que el rendimiento del sistema operativo se degrade) o para el propio rendimiento del sistema operativo (si se quiere evitar el requisito de una memoria principal mucho mayor).

El algoritmo del reloj también se ha comparado con estos otros algoritmos cuando la reserva de marcos es variable y se aplican ámbitos de reemplazamiento tanto global como local (*véase* la siguiente explicación relativa a las políticas de reemplazo) [CARR81, CARR 84]. El algoritmo del reloj se encuentra muy próximo en rendimiento al LRU.

El algoritmo del reloj puede hacerse más potente incrementando el número de bits que utiliza[7]. En todos los procesadores que soportan paginación, se asocia un bit de modificado a cada una de las páginas de la memoria principal y por tanto con cada marco de la memoria principal. Este bit es necesario debido a que, cuando una página se ha modificado, no se la puede reemplazar hasta que se haya escrito de nuevo a la memoria secundaria. Podemos sacar provecho de este bit en el algoritmo del reloj de la siguiente manera. Si tenemos en cuenta los bits de usado y modificado, cada marco de página cae en una de las cuatro categorías siguientes:

---

[7] Por otro lado, si se reduce el número de bits utilizados a 0, el algoritmo del reloj degenera a uno de tipo FIFO.

- No se ha accedido recientemente, no modificada ($u = 0$; $m = 0$)
- Accedida recientemente, no modificada ($u = 1$; $m = 0$)
- No se ha accedido recientemente, modificada ($u = 0$; $m = 1$)
- Accedida recientemente, modificada ($u = 1$; $m = 1$)

Con esta clasificación, el algoritmo del reloj puede actuar de la siguiente manera:

1. Comenzando por la posición actual del puntero, recorremos el *buffer* de marcos. Durante el recorrido, no se hace ningún cambio en el bit de usado. El primer marco que se encuentre con ($u = 0$; $m = 0$) se selecciona para reemplazo.
2. Si el paso 1 falla, se recorre el *buffer* de nuevo, buscando un marco con ($u = 0$; $m = 1$). El primer marco que se encuentre se seleccionará para reemplazo. Durante el recorrido, se pondrá el bit de usado a 0 en cada uno de los marcos que se vayan saltando.
3. Si el paso 2 también falla, el puntero debe haber vuelto a la posición original y todo los marcos en el conjunto tendrán el bit de usado a 0. Se repite el paso 1 y, si resulta necesario el paso 2. Esta vez, se encontrará un marco para reemplazo.

En resumen, el algoritmo de reemplazo de páginas da vueltas a través de todas las páginas del *buffer* buscando una que no se haya modificado desde que se ha traído y que no haya sido accedida recientemente. Esta página es una buena opción para reemplazo y tiene la ventaja que, debido a que no se ha modificado, no necesita escribirse de nuevo en la memoria secundaria. Si no se encuentra una página candidata en la primera vuelta, el algoritmo da una segunda vuelta al *buffer*, buscando una página modificada que no se haya accedido recientemente. Incluso aunque esta página tenga que escribirse antes de ser remplazada, debido al principio de proximidad de referencia, puede no necesitarse de nuevo en el futuro próximo. Si esta segunda pasada falla, todos los marcos en el *buffer* se encuentran marcados como si no hubiesen sido accedidos recientemente y se realiza una tercera pasada.

Esta estrategia se ha utilizado en el esquema de memoria virtual de las versiones antiguas de los Macintosh [GOLD89], mostrados en la Figura 8.18. La ventaja de este algoritmo sobre el algoritmo del reloj básico es que se les otorga preferencia para el reemplazo a las páginas que no se han modificado. Debido a que la página que se ha modificado debe escribirse antes del reemplazo, hay un ahorro de tiempo inmediato.

***Buffering* páginas**. A pesar de que las políticas LRU y del reloj son superiores a FIFO, ambas incluyen una complejidad y una sobrecarga que FIFO no sufre. Adicionalmente, existe el aspecto relativo a que el coste de reemplazo de una página que se ha modificado es superior al de una que no lo ha sido, debido a que la primera debe escribirse en la memoria secundaria.

Una estrategia interesante que puede mejorar el rendimiento de la paginación y que permite el uso de una política de reemplazo de páginas sencilla es el *buffering* de páginas. La estrategia más representativa de este tipo es la usaba por VAX VMS. El algoritmo de reemplazo de páginas es el FIFO sencillo. Para mejorar el rendimiento, una página remplazada no se pierde sino que se asigna a una de las dos siguientes listas: la lista de páginas libres si la página no se ha modificado, o la lista de páginas modificadas si lo ha sido. *Véase* que la página no se mueve físicamente de la memoria; al contrario, la entrada en la tabla de páginas para esta página se elimina y se coloca bien en la lista de páginas libres o bien en la lista de páginas modificadas.

La lista de páginas libres es una lista de marcos de páginas disponibles para lectura de nuevas páginas. VMS intenta mantener un pequeño número de marcos libres en todo momento. Cuando una

página se va a leer, se utiliza el marco de página en la cabeza de esta lista, eliminando la página que estaba. Cuando se va a reemplazar una página que no se ha modificado, se mantiene en la memoria ese marco de página y se añade al final de la lista de páginas libres. De forma similar, cuando una página modificada se va a escribir y reemplazar, su marco de página se añade al final de la lista de páginas modificadas.

El aspecto más importante de estas maniobras es que la página que se va a reemplazar se mantiene en la memoria. De forma que si el proceso hace referencia a esa página, se devuelve al conjunto residente del proceso con un bajo coste. En efecto, la lista de páginas modificadas y libres actúa como una *cache* de páginas. La lista de páginas modificadas tiene también otra función útil: las páginas modificadas se escriben en grupos en lugar de una a una. Esto reduce de forma significativa el número de operaciones de E/S y por tanto el tiempo de acceso disco.

Una versión simple de esta estrategia de *buffering* de páginas la implementa el sistema operativo Mach [RASH88]. En este caso, no realiza distinción entre las páginas modificadas y no modificadas.

Política de reemplazo y tamaño de la *cache*. Como se ha comentado anteriormente, cuando el tamaño de la memoria principal crece, la proximidad de referencia de las aplicaciones va a decrecer. En compensación, los tamaños de las caches pueden ir aumentando. Actualmente, grandes tamaños de *caches*, incluso de varios megabytes, son alternativas de diseño abordables [BORG90]. Con una

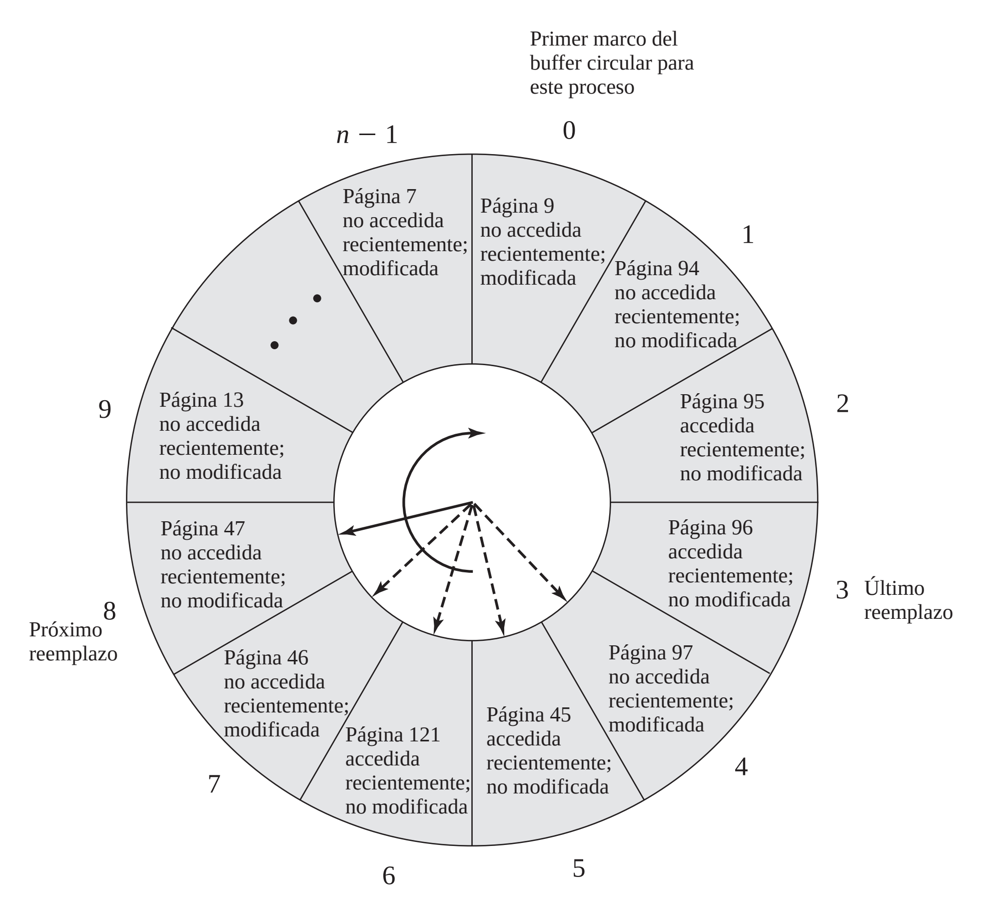


Figura 8.18. El algoritmo de reemplazo de páginas del reloj [GOLD89].

cache de gran tamaño, el reemplazo de páginas de la memoria virtual puede tener un impacto importante en el rendimiento. Si el marco de página destinado al reemplazo está en *cache*, entonces el bloque de *cache* se pierde al mismo tiempo que la página que lo contiene.

En sistemas que utilizan algún tipo de *buffering* de páginas, se puede mejorar el rendimiento de la cache añadiendo a la política de reemplazo de páginas una política de ubicación de páginas en el *buffer* de páginas. La mayoría de sistemas operativos ubican las páginas seleccionando un marco procedente del *buffer* de páginas, de forma arbitraria; utilizando habitualmente una disciplina de tipo FIFO. El estudio reportado en [KESS92] muestra que una estrategia de reemplazo de páginas cuidadosa puede significar entre un 10 y un 20% menos de fallos de *cache* si se compara con un reemplazo simple.

En [KESS92] se examinan diferentes algoritmos de reemplazo de páginas. Estos detalles están más allá del ámbito de este libro, debido a que dependen de detalles de la estructura de las *caches* y sus políticas. La esencia de estas estrategias consiste en traer páginas consecutivas a la memoria principal de forma que se minimice el número de marcos de página que se encuentran proyectados en las mismas ranuras de la cache.

GESTIÓN DEL CONJUNTO RESIDENTE

Tamaño del conjunto residente. Con la memoria virtual paginada, no es necesario, y en algunos casos no es ni siquiera posible, traer todas las páginas de un proceso a la memoria principal para preparar su ejecución. Debido a que el sistema operativo debería saber cuántas páginas traerse, esto es, cuánta memoria principal debería reservar para un proceso en particular. Diferentes factores entran en juego:

- Cuanto menor es la cantidad de memoria reservada para un proceso, mayor es el número de procesos que pueden residir en la memoria principal a la vez. Esto aumenta la probabilidad de que el sistema operativo pueda encontrar al menos un proceso listo para ejecutar en un instante dado, así por tanto, reduce el tiempo perdido debido al *swapping*.
- Si el conjunto de páginas de un proceso que están en memoria es relativamente pequeño, entonces, en virtud del principio de proximidad de referencia, la posibilidad de un fallo de página es mayor (*véase* Figura 8.11.b).
- Más allá de un determinado tamaño, la reserva de más memoria principal para un determinado proceso no tendrá un efecto apreciable sobre la tasa de fallos de página de dicho proceso, debido al principio de proximidad de referencia.

Teniendo en cuenta estos factores, se pueden encontrar dos tipos de políticas existentes en los sistemas operativos contemporáneos. La política de **asignación fija** proporciona un número fijo de marcos de memoria principal disponibles para ejecución. Este número se decide en el momento de la carga inicial de proceso (instante de creación del proceso) y se puede determinar en base al tipo de proceso (interactivo, por lotes, tipo de aplicación) o se puede basar en las guías proporcionadas por el programador o el administrador del sistema. Con la política de asignación fija, siempre que se produzca un fallo de página del proceso en ejecución, la página que se necesite reemplazará una de las páginas del proceso.

Una política de **asignación variable** permite que se reserven un número de marcos por proceso que puede variar a lo largo del tiempo de vida del mismo. De forma ideal, a un proceso que esté causando una tasa de fallos de página relativamente alta de forma continua, indicativo de que el principio proximidad de referencia sólo se aplica de una forma relativamente débil para este proceso, se le otorgarán marcos de página adicionales para reducir esta tasa de fallos; mientras tanto, a un proceso

con una tasa de fallos de páginas excepcionalmente baja, indicativo de que el proceso sigue un comportamiento bien ajustado al principio de proximidad de referencia, se reducirán los marcos reservados, con esperanza de que esto no incremente de forma apreciable la tasa de fallos. El uso de políticas de asignación variable se basa en el concepto de ámbito de reemplazo, como se explicará en la siguiente subsección.

La política de asignación variable podría parecer más potente. Sin embargo, las dificultades de esta estrategia se deben a que el sistema operativo debe saber cuál es el comportamiento del proceso activo. Esto requiere, de forma inevitable, una sobrecarga software por parte del sistema operativo y depende de los mecanismos hardware proporcionados por la propia plataforma del procesador.

Ámbito de reemplazo. La estrategia del ámbito de reemplazo se puede clasificar en global y local. Ambos tipos de políticas se activan por medio de un fallo de página cuando no existen marcos de página libres. Una política de **reemplazo local** selecciona únicamente entre las páginas residentes del proceso que ha generado el fallo de página. Para la identificación de la página a reemplazar en una política de **reemplazo global** se consideran todas las páginas en la memoria principal que no se encuentren bloqueadas como candidatos para el reemplazo, independientemente de a qué proceso pertenezca cada página en particular. Mientras que las políticas locales son más fáciles de analizar, no existe ninguna evidencia convincente de que proporcionen un rendimiento mejor que las políticas globales, que son más atractivas debido a la simplicidad de su implementación con una sobrecarga mínima [CARR84, MAEK87].

Existe una correlación entre el ámbito de reemplazo y el tamaño del conjunto residente (Tabla 8.4). Un conjunto residente fijo implica automáticamente una política de reemplazo local: para mantener el tamaño de conjunto residente, al reemplazar una página se debe eliminar de la memoria principal otra del mismo proceso. Una política de asignación variable puede emplear claramente la política de reemplazo global: el reemplazo de una página de un proceso en la memoria principal por la de otro causa que la asignación de memoria para un proceso crezca en una página mientras que disminuye la del otro. Se verá también que la asignación variable y el reemplazo local son una combinación válida. Se examinan ahora estas tres combinaciones.

Asignación fija, ámbito local. En este caso, se parte de un proceso que se encuentra en ejecución en la memoria principal con un número de marcos fijo. Cuando se da un fallo de página, el sistema operativo debe elegir una página entre las residentes del proceso actual para realizar el reemplazo. Se utilizarían los algoritmos de reemplazo que se han visto en la sección precedente.

Con la política de asignación fija, es necesario decidir por adelantado la cantidad de espacio reservado para un proceso determinado. Esto se puede hacer en base al tipo de aplicación y al tamaño del programa. Las desventajas de esta estrategia son de dos tipos: si las reservas resultan ser demasiado pequeñas, va a haber una alta tasa de fallos de página, haciendo que el sistema multiprogramado completo se ralentice. Si las reservas, por contra, resultan demasiado grandes, habrá muy pocos programas en la memoria principal y habrá mucho tiempo del procesador ocioso o mucho tiempo perdido en *swapping*.

Asignación variable, ámbito global. Esta combinación es, probablemente, la más sencilla de implementar y ha sido adoptada por un gran número de sistemas operativos. En un momento determinado, existen un número de procesos determinado en la memoria principal, cada uno de los cuales tiene una serie de marcos asignados. Normalmente, el sistema operativo también mantiene una lista de marcos libres. Cuando sucede un fallo de página, se añade un marco libre al conjunto residente de un proceso y se trae la página a dicho marco. De esta forma, un proceso que sufra diversos fallos de página crecerá gradualmente en tamaño, lo cual debería reducir la tasa de fallos de página global del sistema.

Table 8.4. Gestión del conjunto residente.

	Reemplazo Local	Reemplazo Global
Asignación Fija	• El número de marcos asignados a un proceso es fijo. • Las páginas que se van a reemplazar se eligen entre los marcos asignados al proceso.	• No es posible
Asignación Variable	• El número de marcos asignados a un proceso pueden variarse de cuando en cuando. • Las páginas que se van a reemplazar se eligen entre los marcos asignados al proceso.	• Las páginas que se van a reemplazar se eligen entre todos los marcos de la memoria principal esto hace que el tamaño del conjunto residente de los procesos varíe.

La dificultad de esta estrategia se encuentra en la elección de los reemplazos cuando no existen marcos libres disponibles, el sistema operativo debe elegir una página que actualmente se encuentra en la memoria para reemplazarla. Esta selección se lleva a cabo entre todos los marcos que se encuentran en la memoria principal, a excepción de los marcos bloqueados como son los usados por el núcleo. Utilizando cualquiera de las políticas vistas en la sección anterior, se tiene que la página seleccionada para reemplazo puede pertenecer a cualquiera de los procesos residentes; no existe ninguna disciplina predeterminada que indique qué proceso debe perder una página de su conjunto residente. Así pues, el proceso que sufre la reducción del tamaño de su conjunto residente no tiene porqué ser el óptimo.

Una forma de encontrar una contraprestación a los problemas de rendimiento potenciales de la asignación variable con reemplazo de ámbito global, se centran en el uso de *buffering* de páginas. De esta forma, la selección de una página para que se reemplace no es tan significativa, debido a que la página se puede reclamar si se hace referencia antes de que un nuevo bloque de páginas se sobrescriba.

Asignación variable, ámbito local. La asignación variable con reemplazo de ámbito local intenta resolver los problemas de la estrategia de ámbito global. Se puede resumir en lo siguiente:

1. Cuando se carga un nuevo proceso en la memoria principal, se le asignan un cierto número de marcos de página a su conjunto residente, basando en el tipo de aplicación, solicitudes del programa, u otros criterios. Para cubrir esta reserva se utilizará la paginación adelantada o la paginación por demanda.

2. Cuando ocurra un fallo página, la página que se seleccionará para reemplazar pertenecerá al conjunto residente del proceso que causó el fallo.

3. De vez en cuando, se reevaluará la asignación proporcionada a cada proceso, incrementándose o reduciéndose para mejorar al rendimiento.

Con esta estrategia, las decisiones relativas a aumentar o disminuir el tamaño del conjunto residente se toman de forma más meditada y se harán contando con los indicios sobre posibles demandas futuras de los procesos que se encuentran activos. Debido a la forma en la que se realiza esta valoración, esta estrategia es mucho más compleja que la política de reemplazo global simple. Sin embargo, puede llevar a un mejor rendimiento.

Los elementos clave en la estrategia de asignación variable con ámbito local son los criterios que se utilizan para determinar el tamaño del conjunto residente y la periodicidad de estos cambios. Una

estrategia específica que ha atraído mucha atención en la literatura es la denominada **estrategia del conjunto de trabajo**. A pesar de que la estrategia del conjunto de trabajo pura sería difícil implementar, es muy útil examinarla como referencia para las comparativas.

El conjunto de trabajo es un concepto acuñado y popularizado por Denning [DENN68, DENN70, DENN80b]; y ha tenido un profundo impacto en el diseño de la gestión de la memoria virtual. El conjunto de trabajo con parámetro Δ para un proceso en el tiempo virtual t, $W(t, \Delta)$ es el conjunto de páginas del proceso a las que se ha referenciado en las últimas Δ unidades de tiempo virtual.

El tiempo virtual se define de la siguiente manera. Considérese la secuencia de referencias a memoria, $r(1), r(2), \ldots$, en las cuales $r(i)$ es la página que contiene la i-esima dirección virtual generada por dicho proceso. El tiempo se mide en referencias a memoria; así $t=1,2,3\ldots$ mide el tiempo virtual interno del proceso.

Se considera que cada una de las dos variables de W. La variable Δ es la ventana de tiempo virtual a través de la cual se observa al proceso. El tamaño del conjunto trabajo será una función nunca decreciente del tamaño de ventana. El resultado que se muestra en la Figura 8.19 (en base a [BACH86]), demuestra la secuencia de referencias a páginas para un proceso. Los puntos indican las unidades de tiempo en las cuales el conjunto de trabajo no cambia. Nótese que para mayor tamaño ventana, el tamaño del conjunto trabajo también es mayor. Esto se puede expresar en la siguiente relación:

$$W(t, \Delta+1) \supseteq W(t, \Delta)$$

El conjunto trabajo es también una función del tiempo. Si un proceso ejecuta durante Δ unidades de tiempo, y terminando el tiempo utiliza una única página, entonces $|W(t, \Delta)|=1$. Un conjunto trabajo también puede crecer hasta llegar a las N páginas del proceso si se accede rápidamente a muchas páginas diferentes y si el tamaño de la ventana lo permite. De esta forma,

$$1 \leq |W(t, \Delta)| \leq min(\Delta,N)$$

La Figura 8.20 indica cómo puede variar el tamaño del conjunto de trabajo a lo largo del tiempo para un valor determinado de Δ. Para muchos programas, los periodos relativamente estables del tamaño de su conjunto de trabajo se alternan con periodos de cambio rápido. Cuando un proceso comienza a ejecutar, de forma gradual construye su conjunto de trabajo a medida que hace referencias a nuevas páginas. Esporádicamente, debido al principio de proximidad de referencia, el proceso deberá estabilizarse sobre un conjunto determinado de páginas. Los periodos transitorios posteriores reflejan el cambio del programa a una nueva región de referencia. Durante la fase de transición algunas páginas del antiguo conjunto de referencia permanecerán dentro de la ventana, Δ, causando un rápido incremento del tamaño del conjunto de trabajo a medida que se van referenciando nuevas páginas. A medida que la ventana se desplaza de estas referencias, el tamaño del conjunto de trabajo se reduce hasta que contiene únicamente aquellas páginas de la nueva región de referencia.

Este concepto del conjunto de trabajo se puede usar para crear la estrategia del tamaño conjunto residente:

1. Monitorizando el conjunto de trabajo de cada proceso.
2. Eliminando periódicamente del conjunto residente aquellas páginas que no se encuentran en el conjunto de trabajo, esto en esencia es una política LRU.
3. Un proceso puede ejecutar sólo si su conjunto trabajo se encuentra en la memoria principal (por ejemplo, si su conjunto residente incluye su conjunto de trabajo).

Secuencia de referencias a páginas	Tamaño de ventana, Δ 2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	15 18 23	15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figura 8.19. Conjunto de trabajo de un proceso definido por el tamaño de ventana.

Esta estrategia funciona debido a que parte de un principio aceptado, el principio de proximidad de referencia, y lo explota para conseguir una estrategia de la gestión de la memoria que minimice los fallos de página. Desgraciadamente, persisten varios problemas en la estrategia del conjunto trabajo:

1. El pasado no siempre predice el futuro. Tanto el tamaño como la pertenencia al conjunto de trabajo cambian a lo largo del tiempo (por ejemplo, *véase* la Figura 8.20).
2. Una medición verdadera del conjunto de trabajo para cada proceso no es practicable. Sería necesario, por cada proceso, asignar un sello de tiempo a cada referencia a una página que asigne el tiempo virtual del proceso y que mantenga una lista ordenada por tiempo de las páginas de cada uno de ellos.
3. El valor óptimo de Δ es desconocido y en cualquier caso puede variar.

Sin embargo el espíritu de esta estrategia sí es válido, el gran número de sistemas operativos intentan aproximarse a la estrategia del conjunto de trabajo. La forma de hacer esto es centrarse no en las referencias exactas a páginas y sí en la tasa de fallos de página. Como se ilustra en la Figura 8.11b, la tasa de fallos de páginas cae a medida que se incrementa el tamaño del conjunto residente de un proceso. El tamaño del conjunto de trabajo debe caer en un punto de esta curva, indicado por *W* en la figura. Por tanto, en lugar de monitorizar el tamaño del conjunto trabajo de forma directa se pueden alcanzar resultados comparables monitorizando la tasa de fallos de página. El razonamiento es el siguiente: si la tasa

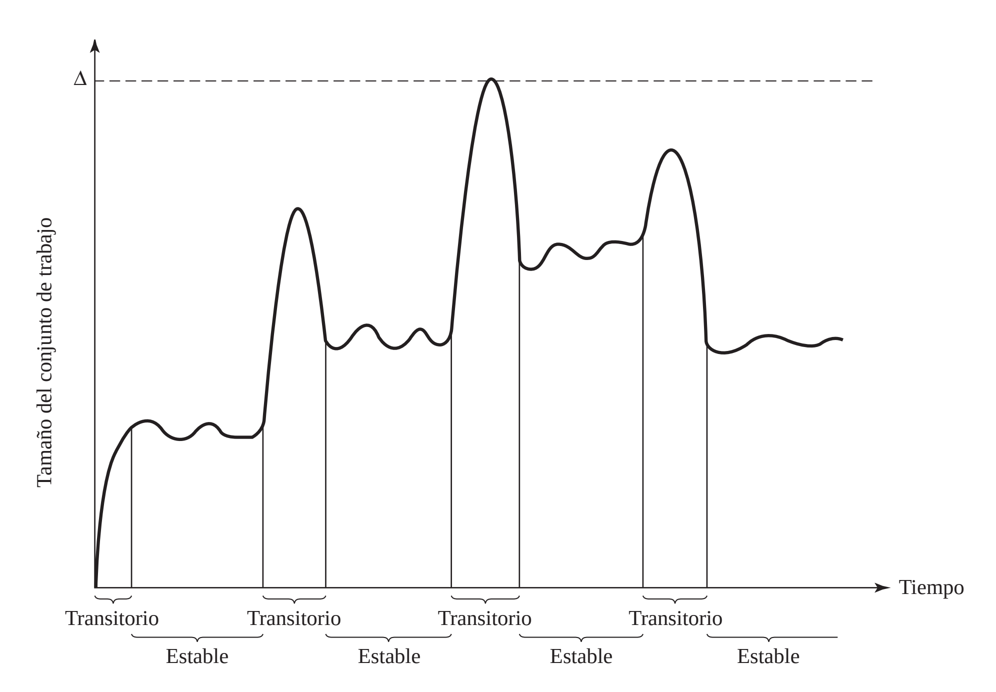


Figura 8.20. Gráfico típico del tamaño del conjunto de trabajo [MAEK87].

de fallos de páginas de un proceso está por debajo de un determinado límite, el sistema de forma global se puede beneficiar de asignar un tamaño de conjunto residente menor para este proceso (debido a que se dispone de un mayor número de marcos de páginas libres para otros procesos) sin dañar al proceso en cuestión (causando un incremento de sus fallos de página). Si la tasa de fallos de página entonces está por encima de un umbral máximo, el proceso puede beneficiarse de un incremento en el tamaño de su conjunto residente (y que así se produzca un menor número de fallos) sin degradar el sistema.

Un algoritmo que sigue esta estrategia es el algoritmo de **frecuencia de fallos de página** (*page fault frequency* – PFF) [CHU72, GUPT78]. El algoritmo necesita un bit de usado que se encuentre asociado a cada página de memoria. Este bit se pondrá a 1 cuando se haya accedido a la página. Cuando se produce un fallo de página, el sistema operativo anotará el tiempo virtual desde el último fallo de página para dicho proceso; esto se puede realizar manteniendo un contador de las referencias a páginas. Se fija un umbral *F*. Si la diferencia de tiempo con el último fallo de página es menor que éste, entonces se añade una página al conjunto residente del proceso. En otro caso, se descartan todas los páginas con el bit de usado a 0, y se reduce el tamaño del conjunto residente de forma acorde. Mientras tanto, se pone a 0 el bit de usado del resto de las páginas. Esta estrategia se puede refinar usando dos umbrales: un umbral máximo que se utiliza para disparar el crecimiento del conjunto residente, y un límite inferior que se utiliza para disparar la reducción de tamaño del conjunto residente.

El tiempo entre fallos de página es recíproco a la tasa de fallos de página. A pesar de ello parecería mejor mantener una medida a lo largo de la ejecución de la tasa de fallos de página, sin embargo la utilización de una medida de tiempo sencilla es un compromiso razonable que permite que las decisiones relativas al tamaño del conjunto residente se basen en la tasa de fallos de página. Si una estrategia de este estilo se complementa con el *buffering* de páginas, se puede conseguir como resultado un rendimiento bastante bueno.

Sin embargo, existe un fallo grave en la estrategia adoptada por PFF, que hace que su comportamiento no sea bueno durante los periodos transitorios cuando se produce un desplazamiento hacia

una nueva región de referencia. Con PFF ninguna página sale del conjunto residente antes de que hayan pasado *F* unidades de tiempo virtual desde su última referencia. Durante estos periodos entre dos regiones de referencia, la rápida sucesión de fallos de página hace que el conjunto residente del proceso crezca antes de que las páginas de la antigua región de referencia se expulsen; los súbitos picos en las solicitudes de memoria pueden producir desactivaciones y reactivaciones de procesos innecesarias, que se corresponden con cambios de proceso y sobrecargas de *swapping* no deseables.

Una estrategia que intenta manejar este fenómeno de transición entre regiones de referencia con una sobrecarga relativamente baja comparado con PFF es la política de **conjunto de trabajo con muestreo sobre intervalos variables** (*variable-interval sampled working set* – VSWS) [FERR83]. La política VSWS evalúa el conjunto de trabajo del proceso en instantes de muestreo basados en el tiempo virtual transcurrido. Al comienzo del intervalo de muestreo, los bits de usado de las páginas residentes de procesos se ponen a 0; al final, sólo las páginas a las que se ha hecho referencia durante el intervalo mantendrán dicho bit a 1; estas páginas se mantienen dentro del conjunto residente del proceso a lo largo del siguiente intervalo, mientras que las otras se descartan. De esta forma el tamaño del conjunto residente solamente decrecerá al final del intervalo. Durante cada intervalo, todas las páginas que han causado fallo se añaden al conjunto residente; de esta forma el conjunto residente mantiene un tamaño fijo o crece durante el intervalo.

La política VSWS toma tres diferentes parámetros:

M: la duración mínima del intervalo de muestreo

L: la duración máxima del intervalo de muestreo

Q: el número de fallos de página que se permite que ocurran entre dos instantes de muestreo

La política VSWS es la siguiente:

1. Si el tiempo virtual entre el último muestreo alcanza *L*, se suspende el proceso y se analizan los bits de usado.
2. Si, antes de que el tiempo virtual transcurrido llegue a *L*, ocurren *Q* fallos de página,
   a) Si el tiempo virtual desde el último muestreo es menor que *M*, se espera hasta que el tiempo virtual transcurrido alcance dicho valor para suspender el proceso y analizar los bits de usado.
   b) Si el tiempo virtual desde el último muestreo es mayor o igual a *M*, se suspende el proceso y se analizan sus bits de usado.

Los valores de los parámetros se toman de forma que el muestreo se dispare habitualmente cuando ocurre el fallo de página *Q* después del último muestreo (caso 2b). Los otros dos parámetros (*M* y *L*) proporcionan fronteras de protección para condiciones excepcionales. La política VSWS intenta reducir el pico de solicitudes de memoria causadas por una transición abrupta entre dos áreas de referencia, incrementando para ello la frecuencia de muestreo, y por tanto la tasa a la cual las páginas no utilizadas se descartan del conjunto residente, cuando la tasa de fallos de página, se incrementa. La experiencia con esta técnica en el sistema operativo del *mainframe* de Bull, GCOS 8, indica que este mecanismo es tan sencillo de implementar como PFF y mucho más efectivo [PIZZ89].

POLÍTICA DE LIMPIEZA

La política de limpieza es la opuesta a la política de recuperación; se encarga de determinar cuándo una página que está modificada se debe escribir en memoria secundaria. Las dos alternativas más co-

munes son la limpieza bajo demanda y la limpieza adelantada. Con la **limpieza bajo demanda**, una página se escribe a memoria secundaria sólo cuando se ha seleccionado para un reemplazo. En la política de la **limpieza adelantada** se escribe las páginas modificadas antes de que sus marcos de páginas se necesiten, de forma que las páginas se puedan escribir en lotes.

Existe un peligro en perseguir cualquiera de estas dos políticas hasta el extremo. Con la limpieza adelantada, una página se escribe aunque continúe en memoria principal hasta que el algoritmo de reemplazo páginas indique que debe eliminarse. La limpieza adelantada permite que las páginas se escriban en lotes, pero tiene poco sentido escribir cientos o miles de páginas para darnos cuenta de que la mayoría de ellas van a modificarse de nuevo antes de que sean reemplazadas. La capacidad de transferencia de la memoria secundaria es limitada y no se debe malgastar con operaciones de limpieza innecesarias.

Por otro lado, con la limpieza bajo demanda, la escritura de una página modificada colisiona con, y precede a, la lectura de una nueva página. Esta técnica puede minimizar las escrituras de páginas, pero implica que el proceso que ha sufrido un fallo página debe esperar a que se completen dos transferencias de páginas antes de poder desbloquearse. Esto implica una reducción en la utilización del procesador.

Una estrategia más apropiada incorpora *buffering* páginas. Esto permite adoptar la siguiente política: limpiar sólo las páginas que son reemplazables, pero desacoplar las operaciones de limpieza y reemplazo. Con *buffering* de páginas, las páginas reemplazadas pueden ubicarse en dos listas: modificadas y no modificadas. Las páginas en la lista de modificadas pueden escribirse periódicamente por lotes y moverse después a la lista de no modificadas. Una página en la lista de no modificadas puede, o bien ser reclamaba si se referencia, o perderse cuando su marco se asigna a otra página.

CONTROL DE CARGA

El control de carga determina el número de procesos que residirán en la memoria principal, eso se denomina el grado de multiprogramación. La política de control de carga es crítica para una gestión de memoria efectiva. Si hay muy pocos procesos residentes a la vez, habrá muchas ocasiones en las cuales todos los procesos se encuentren bloqueados, y gran parte del tiempo se gastarán realizando *swapping*. Por otro lado, si hay demasiados procesos residentes, entonces, de media, el tamaño de conjunto residente de cada proceso será poco adecuado y se producirán frecuentes fallos de página. El resultado es el trasiego (*thrashing*).

Grado de multiprogramación. El trasiego se muestra en la Figura 8.21. A medida que el nivel de multiprogramación aumenta desde valores muy pequeños, cabría esperar que la utilización del procesador aumente, debido a que hay menos posibilidades de que los procesos residentes se encuentren bloqueados. Sin embargo, se alcanza un punto en el cual el tamaño de conjunto residente promedio no es adecuado. En este punto, el número de fallos de páginas se incrementa de forma dramática, y la utilización del procesador se colapsa.

Existen numerosas formas de abordar este problema. Un algoritmo del conjunto de trabajo o de frecuencia de fallos de página incorporan, de forma implícita, control de carga. Sólo aquellos procesos cuyo conjunto residente es suficientemente grande se les permite ejecutar. En la forma de proporcionar el tamaño de conjunto residente necesario para cada proceso activo, se encuentra la política que automática y dinámicamente determina el número de programas activos.

Otra estrategia, sugerida por Denning [DENN80b], se conoce como el *criterio de* $L = S$, que ajusta el nivel de multiprogramación de forma que el tiempo medio entre fallos de página se iguala al tiempo medio necesario para procesar un fallo de página. Los estudios sobre el rendimiento indican

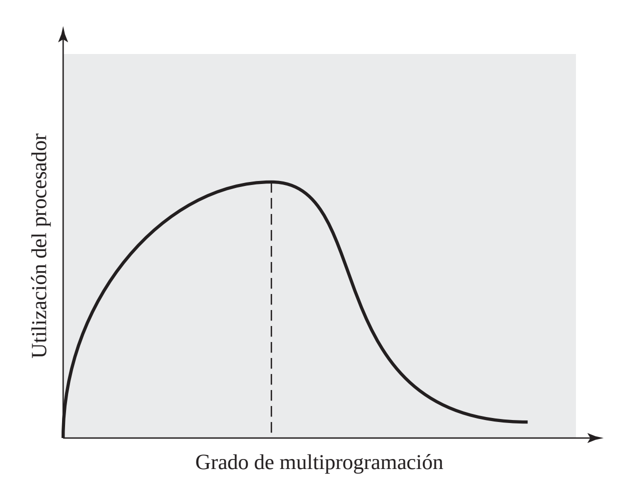


Figura 8.21. Efectos de la multiprogramación.

que éste es el punto en el cual la utilización del procesador es máxima. Una política que presenta un efecto similar, propuesta en [LERO76], es el *criterio del 50%*, que intenta mantener la utilización del dispositivo de paginación aproximadamente al 50%. De nuevo, los estudios sobre el rendimiento indican que éste es el punto para el cual la utilización del procesador es máxima.

Otra alternativa es adaptar el algoritmo de reemplazo de páginas del reloj descrito anteriormente (Figura 8.16). [CARR84] describe la técnica, usando un ámbito global, que implica monitorizar la tasa a la cual el puntero recorre el *buffer* circular de marcos. Si la tasa está por debajo de un nivel de umbral determinado, éste indica alguna (o las dos) circunstancias siguientes:

1. Si están ocurriendo pocos fallos de página, que implican pocas peticiones para avanzar este puntero.
2. Por cada solicitud, el número de marcos promedio que se recorren por el puntero es pequeño, que indica que hay muchas páginas residentes a las que no se hace referencia y que son realmente reemplazables.

En ambos casos, el grado de multiprogramación puede incrementarse con seguridad. Por otro lado, si la tasa de recorrido circular del puntero supera un umbral máximo, indica que la tasa de fallos de página es alta o que hay dificultad para encontrar páginas reemplazables, lo cual implica que el grado de multiprogramación es demasiado alto.

Suspensión de procesos. Si se va a reducir el grado de multiprogramación, uno o más de los procesos actualmente residentes deben suspenderse (enviarlos a *swap*). [CARR84] proporciona seis posibilidades:

- **Procesos con baja prioridad.** Esto implementa una decisión de la política de activación y no se encuentra relacionada con cuestiones de rendimiento.
- **Procesos que provoca muchos fallos.** La razón es que hay una gran probabilidad de que la tarea que causa los fallos no tenga su conjunto de trabajo residente, y el rendimiento sufrirá menos si dicha tarea se suspende. Adicionalmente, esta elección trae una ventaja asociada debido que se bloquea un proceso que estaría a punto de bloquearse de cualquier manera, y que si se elimina se evita por tanto la sobrecarga del reemplazo de páginas y la operación de E/S.

- **Proceso activado hace más tiempo.** Éste es el proceso que tiene menor probabilidad de tener su conjunto de trabajo residente.
- **Proceso con el conjunto residente de menor tamaño.** Éste requerirá un menor esfuerzo para cargarse de nuevo. Sin embargo, penaliza a aquellos programas con una proximidad de referencias muy fuerte.
- **Proceso mayor.** Éste proporciona un mayor número de marcos libres en una memoria que se encuentra sobrecarga, haciendo que futuras desactivaciones sean poco probables a corto plazo.
- **Proceso con la mayor ventana de ejecución restante.** En la mayoría de esquemas de activación de procesos, un proceso sólo puede ejecutarse durante una determinada rodaja de tiempo antes de recibir la interrupción y situarse al final de la lista de Listos. Esta estrategia se aproxima a la disciplina de activación de primero el proceso con menor tiempo en ejecución.

Es como en otras muchas áreas del diseño de sistemas operativos, la selección de la política más apropiada es una cuestión a considerar y depende de muchos otros factores de diseño en el sistema operativo así como de las características de los programas que se ejecutarán.

8.3. GESTIÓN DE MEMORIA DE UNIX Y SOLARIS

Debido a que UNIX pretende ser un sistema independiente de la máquina, su esquema de gestión de memoria variará de un sistema a otro. En las primeras versiones, UNIX utilizaba particionamiento variable sin ningún esquema de memoria virtual. Las implantaciones actuales de UNIX y Solaris utilizan la memoria virtual paginada.

En SVR4 y Solaris, existen dos esquemas de gestión de memoria separados. El **sistema de paginación** proporciona las funcionalidades de la memoria virtual para asignar marcos de página en la memoria principal a los diferentes procesos y también asignar marcos de página a *buffers* de bloques de disco. A pesar de que éste es un esquema de gestión de memoria efectivo para los procesos de usuario y la E/S de disco, un esquema de la memoria virtual paginada es menos apropiado para gestionar la asignación de memoria del núcleo. Por esas cuestiones, se utiliza un **asignador de memoria del núcleo**. Se van a examinar estos dos mecanismos en orden.

SISTEMA DE PAGINACIÓN

Estructuras de datos. Para la memoria virtual paginada, UNIX utiliza varias estructuras de datos que, con pequeñas diferencias, son independientes de la máquina (Figura 8.22 y Tabla 8.5):

- **Tabla de páginas.** Habitualmente, habrá una tabla de páginas por proceso, con una entrada por cada página de memoria virtual de dicho proceso.
- **Descriptor de bloques de disco.** Asociado a cada página del proceso hay una entrada en esta tabla que indica la copia en disco de la página virtual.
- **Tabla de datos de los marcos de página.** Describe cada marco de memoria real y se indexa por medio de un número marco. El algoritmo de reemplazo usa esta tabla.
- **Tabla de utilización de *swap*.** Existe una tabla de uso de *swap* por cada dispositivo de intercambio, con una entrada por cada página de dicho dispositivo.

La mayoría de los campos definidos en la Tabla 8.5 proporcionan su propia descripción. Unos pocos de ellos requieren una explicación adicional. El campo Edad en la entrada de la tabla de pági-

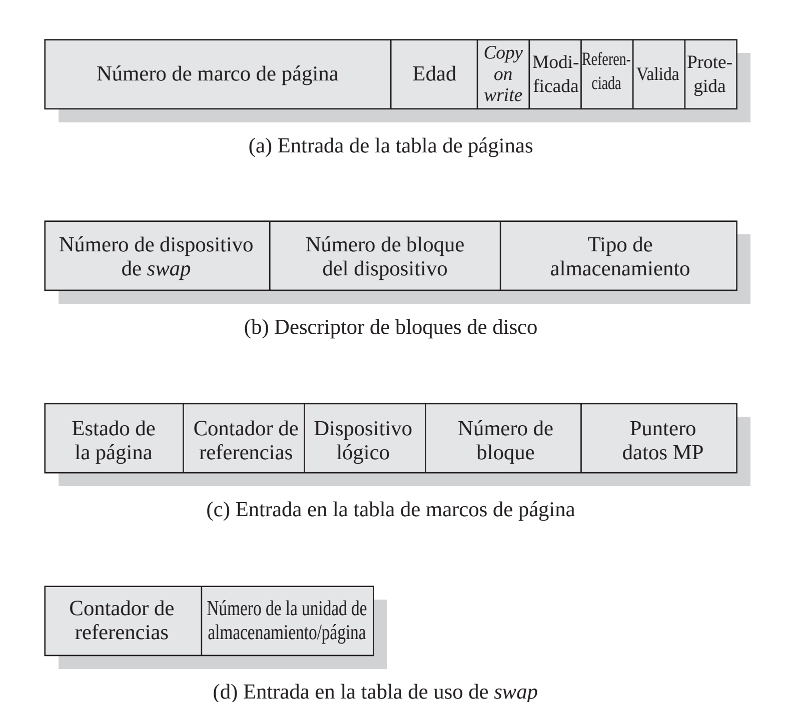


Figura 8.22. Formatos de gestión de memoria de UNIX SVR4.

nas es un indicador de cuánto tiempo hace que el programa no ha hecho referencia a este marco. Sin embargo, el número de bits y la frecuencia de actualización de este campo son dependientes de la implementación. Por tanto, no hay un uso universal de este campo por parte de UNIX para las políticas de reemplazo de páginas.

El campo de Tipo de Almacenamiento en el descriptor de bloques de disco se necesita por la siguiente razón: cuando un fichero ejecutable se usa por primera vez para crear un nuevo proceso, sólo parte del programa y de los datos de dicho fichero se cargan en la memoria real. Más tarde, según se van dando fallos de página, nuevas partes del programa de los datos se irán cargando. Es sólo en el momento de la carga inicial cuando se crean las páginas de memoria virtual y se las asocia con posiciones en uno de los dispositivos que se utiliza para ello. En ese momento, el sistema operativo indica si es necesario limpiar (poner a 0) las posiciones en el marco de página antes de la primera carga de un bloque de programa o datos.

Reemplazo de páginas. La tabla de marcos de página se utiliza para el reemplazo de las páginas. Se usan diferentes punteros para crear listas dentro esta tabla. Todos los marcos disponibles se enlazan en una lista de marcos libres disponibles para traer páginas. Cuando el número de marcos disponibles cae por debajo un determinado nivel, el núcleo quitará varios marcos para compensar.

El algoritmo de reemplazo páginas usado en SVR4 es un refinamiento del algoritmo del reloj (Figura 8.16) conocido como el algoritmo del reloj con dos manecillas (Figura 8.23), el algoritmo utiliza el bit de referencia en la entrada de la tabla de páginas por cada página en memoria que sea susceptible de selección (no bloqueada) para un reemplazo. Este bit se pone a 0 cuando la página se trae por primera vez y se pone a 1 cuando se ha hecho referencia a la página para lectura o escritura. Una de las manecillas del algoritmo del reloj, la manecilla delantera, recorre las páginas de la lista de páginas seleccionables y pone el bit de referencia a 0 para cada una de ellas. Un instante después, la manecilla

trasera recorre la misma lista y verifica el bit de referencia. Si el bit está puesto a 1, entonces se ha hecho referencia a la página desde el momento en que pasó la manecilla delantera por ahí, estos marcos se saltan. Si el bit está a 0, entonces no se ha hecho referencia a dicha página en el intervalo de tiempo de la visita de las dos manecillas; estas páginas se colocan en la lista de páginas expulsables.

Dos parámetros determinan la operación del algoritmo:

- **La tasa de recorrido.** La tasa a la cual las dos manecillas recorren la lista de páginas, en páginas por segundo.
- **La separación entre manecillas.** El espacio entre la manecillas delantera y la trasera.

Estos dos parámetros vienen fijados a unos valores por omisión en el momento de arranque, dependiendo de la cantidad de memoria física. El parámetro tasa de recorrido puede modificarse para responder a cambios en las diferentes condiciones del sistema. El parámetro puede variar linealmente entre los valores de recorrido lento (*slowscan*) y recorrido rápido (*fastscan*) (fijados en el momento de la configuración) dependiendo de que la cantidad de memoria libre variando entre los valores *lotsfree* y *minfree*. En otras palabras, a medida que la memoria libre se reduce, las manecillas del reloj se mueven más rápidamente para liberar más páginas. El parámetro de separación entre manecillas indicado por el espacio entre la manecilla delantera y la trasera, por tanto, junto con la tasa de recorrido, indica la ventana de oportunidad para usar una página antes de que sea descartable por falta de uso.

Asignador de memoria de núcleo. El núcleo, a lo largo de su ejecución, genera y destruye pequeñas tablas y *buffers* con mucha frecuencia, cada uno de los cuales requiere la reserva de memoria dinámica. [VAHA96] muestra varios ejemplos:

- El encaminamiento para traducción de una ruta puede reservar un *buffer* para copiar la ruta desde el espacio usuario.

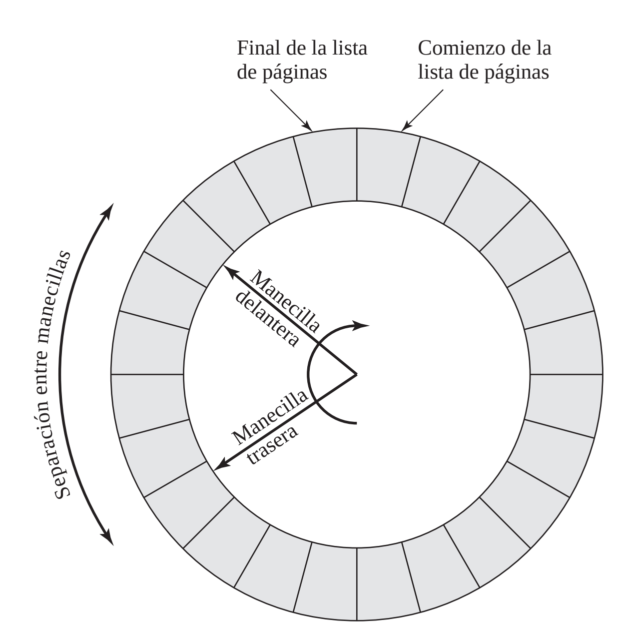


Figura 8.23. Algoritmo de reemplazo de paginas del reloj de dos manecillas.

Tabla 8.5. Parámetros de gestión de memoria de UNIX SVR4.

Entrada de la tabla de páginas

Número de marco de página
Indica el marco de la memoria real.

Edad
Indica cuánto tiempo ha pasado la página en la memoria sin haberse referenciado. La longitud y contenidos de este campo dependen del procesador.

Copy on write[8]
Puesto a 1 cuando más de un proceso comparte esta página. Si uno de los procesos escribe en esta página, se debe hacer primero una copia aparte de la misma para todos los procesos que aún comparten la página original. Esta funcionalidad permite demorar la operación de copia hasta que sea completamente necesaria y evitar los casos en los cuales no llega a serlo.

Modificada
Indica si la página se ha modificado.

Referenciada
Indica que se ha hecho referencia a esta página. Este bit se pone a 0 cuando la página se carga por primera vez y se puede reiniciar periódicamente por parte del algoritmo de reemplazo de páginas.

Válida
Indica si la página se encuentra en la memoria principal.

Protegida
Indica si se permite la operación escritura o no.

Descriptor de bloques de disco

Número de dispositivo de *swap*
Número de dispositivo lógico de almacenamiento secundario que almacena la página correspondiente. Se permite que existan más de un dispositivo para ser utilizados como *swap*.

Número de bloque del dispositivo
Posición del bloque de la página en el dispositivo de *swap*.

Tipo de almacenamiento
El dispositivo de almacenamiento puede ser una unidad de *swap* o un fichero ejecutable. En el último caso, existe una indicación que denota si la memoria virtual asignada debe borrarse o no.

Entrada en la tabla de marcos de página

Estado de la página
Indica si este marco se encuentra disponible o si está asociado a una página. En este último caso, especifica si la página se encuentra en dispositivo de *swap*, en fichero ejecutable, o en una operación de E/S.

Contador de referencias
Número de procesos que hacen referencia a esta página.

Dispositivo lógico
Dispositivo lógico que contiene una copia de la página.

Número de bloque
Posición del bloque de la copia de la página sobre el dispositivo lógico.

Punteros datos MP
Puntero a otros datos que apuntan a otras entradas en la tabla de marcos de página. Con este puntero se puede construir la lista de páginas libres o una lista *hash* de páginas.

Entrada en la tabla de uso de *swap*

Contador de referencias
Número de entradas en la tabla de página que apunta a esta página en el dispositivo de *swap*.

Número de la unidad de almacenamiento/página
Identificador de la página en la unidad almacenamiento

---

[8] N. del T. El término *copy on write* se ha dejado en inglés puesto que es la forma más habitual de denotarlo, incluso en la literatura en castellano.

- La rutina `allocb()` reserva un *buffer* para flujos de tamaño arbitrario.
- Muchas implementaciones de UNIX reservan estructuras *zombie* para recoger el estado de salida e información de la utilización de recursos sobre procesos finalizados.
- En SVR4 y en Solaris, el núcleo reserva muchos objetos (como estructuras del *proc*, v-nodos, y bloques de descripción de fichero) de forma dinámica y bajo demanda.

La mayoría de sus bloques son significativamente más pequeños que el tamaño típico de una página de la máquina, y por tanto el mecanismo de paginación sería muy deficiente a la hora de reserva de la memoria dinámica del núcleo. Para el caso de SVR4, se utiliza una modificación del sistema *buddy*, descrito en la Sección 7.2.

En los sistemas *buddy*, el coste de reservar y liberar un bloque de la memoria es bajo comparado con las políticas de mejor ajuste y primer ajuste [KNUT97]. De esta forma, en el caso de la gestión de la memoria del núcleo, las operaciones de reserva y liberación se deben realizar lo más rápido posible. La desventaja de los sistemas *buddy* es el tiempo necesario para fragmentar y reagrupar bloques.

Barkley y Lee de AT&T propusieron una variación conocida como sistema *buddy* perezoso [BARK89], y ésta es la técnica adoptada por SVR4. Los autores observaron que UNIX a menudo muestra un comportamiento estable en las solicitudes de memoria de núcleo; esto es, la cantidad de peticiones para bloques de un determinado tamaño varía ligeramente a lo largo del tiempo. Por tanto, si un bloque del tamaño 2^i se libera e inmediatamente se reagrupa con su vecino en un bloque del tamaño 2^{i+1}, la próxima vez que el núcleo solicite un bloque de tamaño 2^i, implicará la necesidad de dividir el bloque de tamaño mayor de nuevo. Para evitar esta agrupación innecesaria y su posterior división, el sistema *buddy* perezoso pospone la reagrupación hasta el momento en el que parezca que resulta necesaria, y en ese momento intenta reagrupar el mayor número de bloques posibles.

El sistema *buddy* perezoso usa los siguientes parámetros:

N_i = número actual de bloques de tamaño 2^i.

A_i = número actual de bloques de tamaño 2^i que se encuentran reservados (ocupados).

G_i = número actual de bloques de tamaño 2^i que están libres globalmente; estos son los bloques que se pueden seleccionar para ser reagrupados; si el vecino de uno de estos bloques se encuentra libre, entonces ambos bloques se pueden agrupar en un bloque libre global de tamaño 2^{i+1}. Los bloques libres (huecos) en un sistema *buddy* estándar pueden considerarse libres globales.

L_i = número actual de bloques de tamaño 2^i que se encuentran libres localmente; estos son los bloques que no se encuentran como seleccionables para su grabación. Incluso si el vecino de dicho bloque se encuentra libre, los dos bloques no se reagrupan. En lugar de eso, los bloques libres locales se mantienen a la espera de una petición futura para un bloque de dicho tamaño.

La siguiente relación se mantiene:

$$N_i = A_i + G_i + L_i$$

En general, el sistema *buddy* perezoso intenta mantener un caudal de bloques libres locales y únicamente solicita la reagrupación si el número de bloques libres locales supera un determinado límite. Si hay muchos bloques libres locales, entonces existe la posibilidad de que falten bloques libres en el siguiente nivel. La mayoría del tiempo, cuando se libera un bloque, la reagrupación no se realiza, de

forma que se minimizan los costes de gestión y de operaciones. Cuando se va a reservar un bloque, no se hace distinción alguna entre bloques libres locales y globales; de la misma forma, esto minimiza la gestión.

El criterio que se utiliza para la reagrupación es que el número de bloques libres locales del tamaño determinado no puede exceder el número de bloques reservados para ese tamaño (por ejemplo, se debe tener $L_i \geq A_i$). Ésta es una guía razonable para evitar el crecimiento de bloques libres locales, en los experimentos llevados a cabo por [BARK89] se confirman los resultados de que este esquema proporciona unas mejoras considerables.

Para implementar este esquema, los autores de definen una variable de demora de la siguiente forma:

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

La Figura 8.24 muestra este algoritmo.

Valor inicial de D_i es igual a 0
después de la operación, el valor de D_i se actualiza de la siguiente manera:

(I) si la siguiente operación es una solicitud de reservar un bloque:
si hay bloques libres, se selecciona uno a reserva
si el bloque seleccionado se encuentra libre de forma local
entonces $D_i := D_i + 2$
sino $D_i := D_i + 1$
en otro caso
primero se toman dos bloques dividiendo un bloque mayor en dos (operación recursiva)
se reserva uno y se marca el otro como libre de forma local
D_i permanece sin cambios (pero se puede cambiar D para otros tamaños de bloque debido a la llamada recursiva)

(II) si la siguiente operación es una liberación de un bloque:
Caso $D_i \geq 2$
se marca como libre de forma local y se libera localmente
$D_i := D_i - 2$
Caso $D_i = 1$
se marca como libre de forma global y se libera globalmente; reagrupación si es posible
$D_i := 0$
Caso $D_i = 0$
se marca como libre de forma global y se libera globalmente; reagrupación si es posible
se selecciona un bloque libre local de tamaño $2i$ y se libera de forma global; reagrupación si es posible
$D_i := 0$

Figura 8.24. Algoritmo del sistema *buddy* perezoso.

8.4. GESTIÓN DE MEMORIA EN LINUX

Linux comparte muchas de las características de los esquemas de gestión de la memoria de otras implementaciones de UNIX pero incorpora sus propias características. Por encima de todo, el esquema de gestión de la memoria de Linux es bastante complejo [DUBE98].

En esta sección, vamos a dar una ligera visión general de dos de los principales aspectos de la gestión de la memoria en Linux: la memoria virtual de los procesos y la asignación de la memoria del núcleo.

MEMORIA VIRTUAL EN LINUX

Direccionamiento de la memoria virtual. Linux usa una estructura de tablas de páginas de tres niveles, consistente en los siguientes tipos de tablas (cada tabla en particular tiene el tamaño de una página):

- **Directorio de páginas.** Un proceso activo tiene un directorio de páginas único que tiene el tamaño de una página. Cada entrada en el directorio de páginas apunta a una página en el directorio intermedio de páginas. El directorio de páginas debe residir en la memoria principal para todo proceso activo.
- **Directorio intermedio de páginas.** El directorio intermedio de páginas se expande a múltiples páginas. Cada entrada en el directorio intermedio páginas apunta a una página que contiene una tabla de páginas.
- **Tabla de páginas.** La tabla de páginas también puede expandirse a múltiples páginas. Cada entrada en la tabla de páginas hace referencia a una página virtual del proceso.

Para utilizar esta estructura de tabla de páginas de tres niveles, una dirección virtual en Linux se puede ver cómo consistente en cuatro campos (Figura 8.25) el campo más a la izquierda (el más significativo) se utiliza como índice en el directorio de páginas. El siguiente campo sirve como índice en el directorio intermedio de páginas. El tercer campo se utiliza para indexar en la tabla de páginas. Y con el cuarto campo se proporciona el desplazamiento dentro de la página de la memoria seleccionada.

La estructura de la tabla de páginas en Linux es independiente de plataforma y se diseñó para acomodarse al procesador Alpha de 64 bits, que proporciona soporte hardware para tres niveles de páginas. Con direcciones de 64 bits, la utilización de únicamente dos niveles de páginas en una arquitectura Alpha resultaría unas tablas de páginas y unos directorios de gran tamaño. La arquitectura Pentium/x86 de 32 bits tiene un sistema hardware de paginación de dos niveles. El software de Linux se acomoda al esquema de dos niveles definiendo el tamaño del directorio intermedio de páginas como 1. Hay que resaltar que todas las referencias a ese nivel extra de indirección se eliminan en la optimización realizada en el momento de la compilación, no durante la ejecución. Por tanto, no hay ninguna sobrecarga de rendimiento por la utilización de un diseño genérico de tres niveles en plataformas que sólo soportan dos niveles.

Reserva de páginas. Para mejorar la eficiencia de la lectura y escritura de páginas en la memoria principal, Linux define un mecanismo para manejar bloques de páginas contiguas que se proyectarán sobre bloques de marcos de página también contiguos. Con este fin, también se utiliza el sistema *buddy*. El núcleo mantiene una lista de marcos de página contiguos por grupos de un tamaño fijo; un grupo puede consistir en 1, 2, 4, 8, 16, o 32 marcos de páginas. A lo largo del uso, las páginas se asignan y liberan de la memoria principal, los grupos disponibles se dividen y se juntan utilizando el algoritmo del sistema *buddy*.

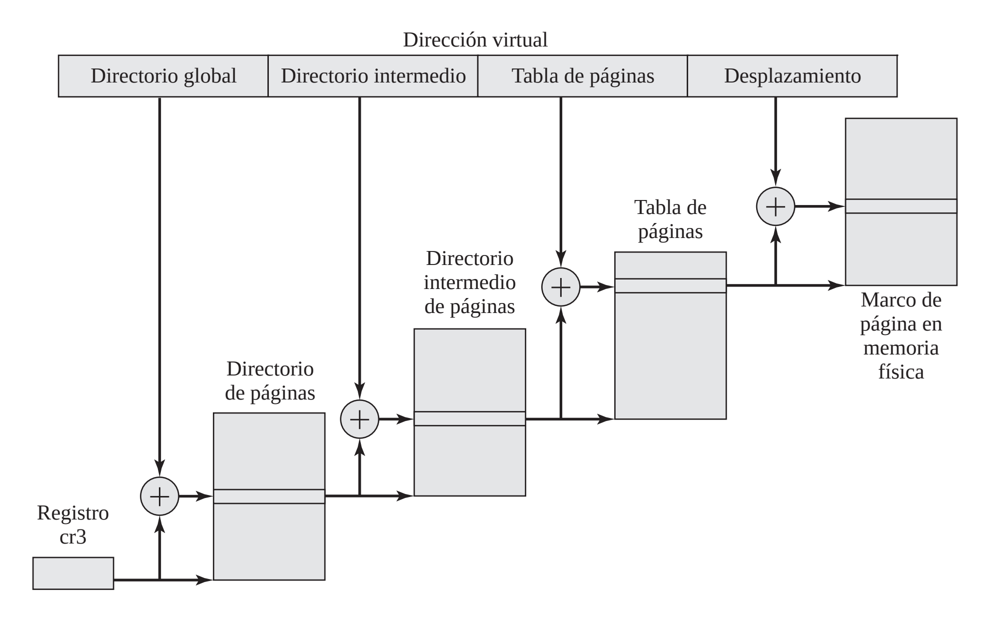


Figura 8.25. Traducción de direcciones en el esquema de memoria virtual de Linux.

Algoritmo de reemplazo de páginas. El algoritmo de reemplazo de páginas de Linux se basaba en el algoritmo del reloj descrito en la Sección 8.2 (*véase* Figura 8.16). En el algoritmo del reloj sencillo, se asocia un bit de usado y otro bit de modificado con cada una de las páginas de memoria principal. En el esquema de Linux, el de usado se reemplaza por una variable de 8 bits. Cada vez que se accede a una página, la variable se incrementa. En segundo plano, Linux recorre periódicamente la lista completa de páginas y decrementa la variable de edad de cada página a medida que va rotando por todas ellas en la memoria principal. Una página con una edad de 0 es una página «vieja» a la que no se ha hecho referencia desde hace algún tiempo y es el mejor candidato para un reemplazo. Cuando el valor de edad es más alto, la frecuencia con la que se ha accedido a la página recientemente es mayor y, por tanto, tiene menor posibilidad de elegirse para un reemplazo de esta forma, el algoritmo de Linux es una variante de la política LRU.

RESERVA DE MEMORIA DEL NÚCLEO

La gestión de la memoria del núcleo se realiza en base a los marcos de página de la memoria principal. Su función básica es asignar y liberar marcos para los diferentes usos. Los posibles propietarios de un marco incluyen procesos en espacio de usuario (por ejemplo, el marco es parte de la memoria virtual de un proceso que se encuentra actualmente residiendo en la memoria real), datos del núcleo reservados dinámicamente, código estático del núcleo, y la cache de páginas[9].

Los fundamentos de la reserva de memoria de núcleo para Linux son los mecanismos de reserva de páginas ya usados para la gestión de la memoria virtual de usuario. Como en el caso del esquema de la memoria virtual, se utiliza el algoritmo *buddy* de forma que la memoria del núcleo

[9] La cache de páginas tiene propiedades similares a un *buffer* de disco, descrito en este capítulo, así como de cache disco, que se verá en el Capítulo 11. Se pospone la discusión sobre la cache de páginas de Linux a dicho Capítulo 11.

se pueda reservar y liberar en unidades de una o más páginas. Debido a que el tamaño mínimo de memoria que se pueda reservar de esta forma es una página, la reserva de páginas únicamente sería insuficiente debido a que el núcleo requiere pequeños fragmentos que se utilizarán durante un corto periodo de tiempo y que son de diferentes tamaños. Para ajustarse a estos pequeños tamaños, Linux utiliza un esquema conocido como *asignación por láminas* (*slab allocation*) [BONW94] dentro una página ya reservada. En una máquina Pentium/x86, el tamaño página es de 4 Kbytes y los fragmentos dentro una página se pueden asignar en tamaños de 32, 64, 128, 252, 508, 2040, y 4080 bytes.

La asignación por láminas es relativamente compleja y no se va a examinar en detalle aquí; una buena descripción se pueda encontrar en [VAHA96]. En esencia, Linux mantiene un conjunto de listas enlazadas, una para cada tamaño del fragmento. Todos los fragmentos se pueden dividir y agregar de una manera similar a la indicada por el algoritmo *buddy*, y también se pueden mover entre las diferentes listas de la forma correspondiente.

8.5. GESTIÓN DE MEMORIA EN WINDOWS

El gestor de memoria virtual en Windows controla la forma en la que se reserva la memoria y cómo se realiza la paginación. El gestor de memoria se ha diseñado para funcionar sobre una variada gama de plataformas y para utilizar tamaños de páginas que van desde los 4 Kbytes hasta los 64 Kbytes. Las plataformas Intel, PowerPC, y MIPS tienen 4096 bytes por página y las plataformas DEC Alpha tienen 8192 bytes por página.

MAPA DE DIRECCIONES VIRTUALES EN WINDOWS

Cada proceso de usuario en Windows puede ver un espacio de direcciones independiente de 32 bits, permitiendo 4 Gbytes de memoria por proceso. Por omisión, una parte de esta memoria se encuentra reservada para el sistema operativo, de forma que en realidad cada usuario dispone de 2 Gbytes de espacio virtual de direcciones disponibles y todos los procesos comparten los mismos 2 Gbytes de espacio de sistema. Existe una opción que permite que el espacio de direcciones crezca hasta los 3 Gbytes, dejando un espacio de sistema de únicamente 1 Gbytes. En la documentación de Windows se indica que esta característica se incluyó para dar soporte a aplicaciones que requieren un uso intensivo de grandes cantidades de memoria en servidores con memorias de varios gigabytes, en los cuales un espacio de direcciones mayor puede mejorar drásticamente el rendimiento de aplicaciones de soporte como la decisión o *data mining*.

La Figura 8.26 muestra el espacio de direcciones virtuales que, por efecto, ve un usuario. Consiste en cuatro regiones:

- 0x00000000 a 0x0000FFFF. Reservada para ayudar a los programadores a capturar asignaciones de punteros nulos.
- 0x00010000 a 0x7FFEFFFF. Espacio de direcciones disponible para el usuario. Este espacio se encuentra dividido en páginas que se pueden cargar de la memoria principal.
- 0x7FFF0000 a 0x7FFFFFFF. Una página de guarda, no accesible por el usuario. Esta página hace que al sistema operativo le resulte más fácil verificar referencias a punteros fuera de rango.
- 0x80000000 a 0xFFFFFFFF. Espacio de direcciones de sistema. Este área de 2 Gbytes se utiliza por parte del Ejecutivo de Windows, el micronúcleo y los manejadores de dispositivos.

PAGINACIÓN EN WINDOWS

Cuando se crea un proceso, puede, en principio, utilizar todo el espacio de usuario de 2 Gbytes (menos 128 Kbytes). Este espacio se encuentra dividido en páginas de tamaño fijo, cualquiera de las cuales se puede cargar en la memoria principal. En la práctica, una página se puede encontrar, a efectos de gestión, en los presentes estados:

- **Disponible.** Páginas que no están actualmente usadas por este proceso.
- **Reservada.** Conjunto de páginas contiguas que el gestor de memoria virtual separa para un proceso pero que no se cuentan para la cuota de memoria usada por dicho proceso. Cuando un proceso necesite escribir en la memoria, parte de esta memoria reservada se asigna al proceso.
- **Asignada.** Las páginas para las cuales el gestor de la memoria virtual ha reservado espacio en el fichero de paginación (por ejemplo, el fichero de disco donde se escribirían las páginas cuando se eliminen de la memoria principal).

La distinción entre la memoria reservada y asignada es muy útil debido a que (1) minimiza la cantidad de espacio de disco que debe guardarse para un proceso en particular, manteniendo espacio libre en disco para otros procesos; y (2) permite que un hilo o un proceso declare una petición de una cantidad de memoria que puede proporcionarse rápidamente si se necesita.

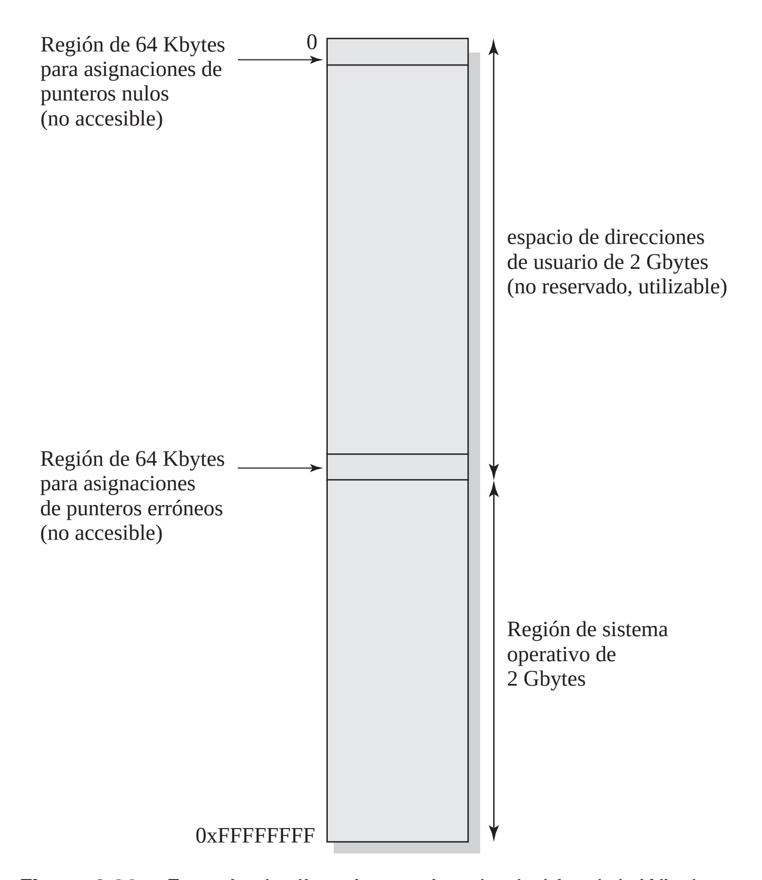


Figura 8.26. Espacio de direcciones virtuales habitual de Windows.

El esquema de gestión del conjunto residente que utiliza Windows es de asignación variable, con ámbito local (*véase* Tabla 8.4). Cuando se activa un proceso por primera vez, se le asigna un cierto número de marcos de página de la memoria principal como conjunto de trabajo. Cuando un proceso hace referencia a una página que no está en la memoria, una de las páginas residentes de dicho proceso se expulsa, trayéndose la nueva página. Los conjuntos de trabajo de los procesos activos se ajustan usando las siguientes condiciones generales:

Cuando hay memoria principal disponible, el gestor de la memoria virtual permite que los conjuntos residentes de los procesos activos crezcan. Para hacer esto, cuando ocurre un fallo página, se trae la nueva página a la memoria sin expulsar una página antigua, haciendo que se incremente el conjunto residente de proceso en una página.

Cuando la memoria empieza escasear, el gestor de la memoria virtual recupera la memoria de sistema moviendo las páginas que se han utilizado hace más tiempo de cada uno de los procesos hacia *swap*, reduciendo el tamaño de esos conjuntos residentes.

8.6. RESUMEN

Para poder usar de forma eficiente el procesador y las funciones de E/S, es aconsejable mantener el mayor número de procesos posibles en la memoria principal. Además, es deseable liberar al programador de las restricciones de tamaño en el desarrollo de programas.

La vía para llevar a cabo ambas recomendaciones es por medio de la memoria virtual. Con la memoria virtual, todas las referencias a direcciones son lógicas y se traducen en tiempo de ejecución a direcciones reales. Esto permite que un proceso se pueda ubicar en cualquier parte de la memoria principal e incluso que dicha ubicación cambie a lo largo del tiempo. La memoria virtual también permite que un proceso se dividida en fragmentos. Estos fragmentos no tienen porqué estar ubicados de forma contigua en la memoria principal durante su ejecución y, en realidad, tampoco se necesita que todos los fragmentos del proceso se encuentren en la memoria principal durante dicha ejecución.

Las dos estrategias básicas para proporcionar memoria virtual son paginación y segmentación. Por medio de la paginación, cada proceso se divide en páginas de tamaño fijo, relativamente pequeñas. La segmentación proporciona la posibilidad de definir fragmentos de tamaño variable. Se pueden combinar segmentación y paginación en un único esquema de la gestión de la memoria.

Un esquema de gestión de la memoria virtual requiere soporte hardware y software. El soporte hardware lo proporciona el procesador. Dicho soporte incluye traducción dinámica de las direcciones virtuales en direcciones físicas y la generación de interrupciones cuando se hace referencia a una página o segmento que no se encuentra en la memoria principal. Dichas interrupciones disparan el software de la gestión de la memoria del sistema operativo.

En este capítulo hemos tratado diversas consideraciones de diseño relativas al soporte del sistema operativo:

- **Política de recuperación.** Las páginas se pueden traer bajo demanda, o con una política de paginación adelantada, que agrupa la carga de varias páginas a la vez.
- **Política de ubicación.** En un sistema de segmentación pura, cuando un segmento se trae a memoria debe ajustarse al espacio en memoria que se encuentra disponible.
- **Política de reemplazo.** Cuando la memoria se encuentra llena, se debe decidir qué página o páginas van a reemplazarse para traer una nueva.
- **Gestión del conjunto reciente.** El sistema operativo debe decidir cuánta memoria asigna a un proceso en particular cuando dicho proceso se trae a la memoria. Esto se puede hacer me-

diante asignación estática, realizada en el momento de crear un proceso, o se puede cambiar dinámicamente.

- **Política de limpieza.** Las páginas del proceso que se han modificado se pueden escribir en el momento de reemplazo, o se puede articular una política de limpieza adelantada, la cual agrupa la actividad de escritura de páginas en varias de ellas a la vez.
- **Control de carga.** El control de carga es el responsable de determinar el número de procesos que van a residir en la memoria principal en un instante determinado.

8.7. LECTURA RECOMENDADA Y PÁGINAS WEB

Como era previsible, la memoria virtual se encuentra tratada en la mayoría de libros sobre sistemas operativos. [MILE92] proporciona un buen resumen de varias áreas de investigación. [CARR84] proporciona un excelente examen en profundidad de las cuestiones de rendimiento. El artículo clásico, [DENN70] es aún una lectura muy recomendada. [DOWD93] proporciona un análisis de rendimiento profundo de varios organismos de reemplazo de páginas. [JACO98a] es una buena revisión de los aspectos de diseño de la memoria virtual; incluye una explicación sobre las tablas de páginas invertidas. [JACO98b] examina la organización del hardware de la memoria virtual para varios microprocesadores.

Una lectura más sobria, [IBM86], que cuenta de forma detallada las herramientas y opciones disponibles para la administración de la optimización de las políticas de la memoria virtual de los sistemas MVS. Este documento ilustra perfectamente la complejidad del problema.

[VAHA96] es uno de los mejores tratamientos de los esquemas de la gestión de la memoria utilizados en varios tipos de sistemas UNIX. [GORM04] es un tratamiento profundo de la gestión de la memoria en Linux.

CARR84 Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press, 1984.

DENN70 Denning, P. «Virtual Memory.» *Computing Surveys.* Septiembre, 1970.

DOWD93 Dowdy, L. y Lowery, C. *P.S. to Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1993.

GORM04 Gorman, M. *Understanding the Linux Virtual Memory Management*. Upper Saddle River, NJ: Prentice Hall, 2004.

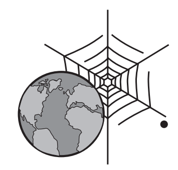
IBM86 IBM Nacional Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook*. Dallas Systems Center Technical Bulletin G320-0597, Junio 1986.

JACO98a Jaboc, B. y Mudge, T. «Virtual Memory: Issues of Implementation.» *Computer*, Junio 1998.

JACO98b Jaboc, B. y Mudge, T. «Virtual Memory in Contemprany Microprocessors» *IEEE Micro*, Agosto 1998.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New Cork: McGraw-Hill, 1992.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.



Páginas Web recomendadas:

- **The Memory Management Reference.** Una buena fuente de documentos y enlaces sobre todo los aspectos de la gestión de la memoria.

8.8. TÉRMINOS CLAVE, CUESTIONES DE REPASO, Y PROBLEMAS

TÉRMINOS CLAVE

asignación por láminas	memoria real	proximidad de referencia
conjunto de trabajo	memoria virtual	segmentación
conjunto reciente	página	segmento
fallo de página	paginación	tabla de páginas
fragmentación externa	paginación adelantada	tabla de segmentos
fragmentación interna	paginación por demanda	tabla *hash*
gestión del conjunto residente	política de recuperación	TLB
hashing	política de reemplazo	traducción asociativa
marco	política de ubicación	trasiego

CUESTIONES DE REPASO

8.1. ¿Cuál es la diferencia entre la paginación sencilla y la paginación con memoria virtual?

8.2. Explique el trasiego o *thrashing*.

8.3. ¿Por qué el principio de proximidad de referencia es crucial para el uso de la memoria virtual?

8.4. ¿Qué elementos se encuentran típicamente en la entrada de tabla de páginas? Defina brevemente cada elemento.

8.5. ¿Cuál es el propósito de la TLB?

8.6. Defina brevemente las alternativas para la política de recuperación.

8.7. ¿Cuál es la diferencia entre la gestión del conjunto residente y la política de reemplazo de páginas?

8.8. ¿Cuál es la relación entre los algoritmos de reemplazo de páginas FIFO y del reloj?

8.9. ¿Cuál es el cometido del *buffering* de páginas?

8.10. ¿Por qué no es posible combinar a la política de reemplazo global y la política de asignación fija?

8.11. ¿Cuál es la diferencia entre el conjunto reciente y el conjunto de trabajo?

8.12. ¿Cuál es la diferencia entre la limpieza por demanda y la limpieza adelantada?

PROBLEMAS

8.1. Suponga que la tabla de páginas del proceso actualmente en ejecución es la siguiente. Todos los números están en formato decimal, todos se encuentran numerados comenzando desde cero, y todas las direcciones son direcciones de memoria a nivel de byte. El tamaño de página es de 1024 bytes.

Número de página virtual	Bit de valida	Bit de referenciada	Bit de modificada	Número de marco de página
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a) Describa exactamente cómo, en términos reales, se generaría a partir de la dirección virtual la dirección de memoria física, por medio de la traducción realizada por la CPU.

b) ¿Qué dirección física, si hay alguna, se corresponderá con las siguientes direcciones virtuales? (No hace falta que trate ningún fallo de página, se produce).

i) 1052

ii) 2221

iii) 5499

8.2. Considere un sistema de memoria virtual paginada con direcciones virtuales de 32 bits y páginas de 1 Kbyte. Cada entrada en la tabla de páginas requiere 32 bits. Y se desea limitar el tamaño de las tablas de páginas a una única página.

a) ¿Cuántos niveles de tablas de páginas se necesitan?

b) ¿Cuál es el tamaño de la tabla de páginas de cada nivel? *Sugerencia*: uno de los tamaños de tabla de página es menor.

c) La tabla de páginas menor se puede usar en la parte más alta o la parte más baja de la jerarquía de tablas de páginas. ¿Cuál de las estrategias implican menor consumo de páginas?

8.3. a) ¿Cuánto espacio de memoria se necesita para las tablas de páginas de usuario de la Figura 8.4?

b) Asumiendo que se quiere implementar una tabla de páginas invertida de tipo *hash* para el mismo esquema de direccionamiento mostrado en la Figura 8.4, por medio de una función *hash* que proyecta el número página de 20 bits en un valor *hash* es de 6 bits. La entrada en la tabla contiene el número de página, el número de marco, y un puntero a la cadena. Si la tabla de páginas dispone de hasta tres entradas adicionales por colisiones por cada entrada *hash*, ¿cuánto espacio de memoria estaría ocupando la tabla de páginas invertida?

8.4. Un proceso tiene cuatro marcos reservados para el uso (los siguientes números están en formato decimal, y todas las numeraciones comienzan desde 0). La siguiente tabla muestra el instante de tiempo en el que se cargó la última página en cada marco, el instante de tiempo del último acceso a cada página, el número de página virtual, los bits de referenciada (R) y modificada (M) para cada uno de los marcos de página (los instantes de tiempo están expresados en *ticks* de reloj desde el comienzo del proceso).

Número de página virtual	Marco de página	Instante de carga	Instante de referencia	Bit R	Bit M
2	0	60	161	0	1
1	1	130	160	1	0
0	2	26	162	1	0
3	3	20	163	1	1

Se produce un fallo de página para la página virtual 4 en el instante 164. ¿Qué marco de página reemplazará su contenido para cada una de las siguientes políticas de la gestión de la memoria? Explique por qué en cada uno de los casos.

a) FIFO

b) LRU

c) Reloj

d) Óptimo (usando la siguiente cadena de referencias)

Tomando el estado de la memoria antes mencionado, justo en el instante en el que se produce el fallo de página, y teniendo en cuenta la siguiente cadena de referencias a páginas virtuales:

4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2

e) ¿Cuántos fallos de página ocurrirían si se usase la política del conjunto de trabajo con LRU tomando una ventana de tamaño 4 en lugar de asignación fija? Indique claramente cuándo se produce cada fallo de página.

8.5. Un proceso hace referencia a cinco páginas, A, B, C, D, y E, en el siguiente orden:

A; B; C; D; A; B; E; A; B; C; D; E

Asuma que el algoritmo de reemplazo es FIFO, y encuentre el número de transferencias de páginas durante la anterior secuencia de referencias que comienza con la memoria principal vacía con una limitación de tres marcos de página. Repita lo mismo para el caso de cuatro marcos de página.

8.6. Un proceso contiene ocho páginas virtuales en disco y se asigna de forma fija cuatro marcos de página de memoria principal. La traza de las páginas es la siguiente:

1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5,1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3

a) Muestre las sucesivas páginas residentes en los cuatro marcos utilizando la política de reemplazo LRU. Calcule el índice de acierto de la memoria principal. Asúmase que los marcos están inicialmente vacíos.

b) Repita el apartado (a) para política de reemplazo FIFO.

c) Compare ambos índices de acierto y comente la efectividad de utilizar FIFO como aproximación a LRU en lo referente a esta traza en particular.

8.7. En los sistemas VAX, las tablas de página de los usuarios se colocan en direcciones virtuales en el espacio de sistema. ¿Cuál es la ventaja de tener las tablas de página de usuarios en la memoria virtual en lugar de la memoria principal? ¿Cuáles son las desventajas?

8.8. Supóngase las siguiente instrucciones de programa

```
for(i=1;i<=n;i++)
    a[i]=b[i]+c[i];
```

Si se ejecutan en una memoria con tamaño de página de 1000 palabras. Siendo $n = 1000$. Usando una máquina que tiene un conjunto completo de instrucciones registro-registro y que utiliza registros índice, escriba un programa hipotético que implemente las siguientes instrucciones. Posteriormente, muestre la secuencia de referencias a páginas durante su ejecución.

8.9. Los sistemas IBM/370 utilizan una estructura de memoria de dos niveles y denominan a cada uno de los niveles como segmentos y páginas, a pesar de que la parte segmentación carece de muchas de las características descritas anteriormente en este capítulo. Para arquitectura básica 370, el tamaño de página puede ser o bien 2 Kbytes o 4 Kbytes, y el tamaño segmento se fija entre 64 Kbytes o 1 Mbyte. Para las arquitecturas 370/XA y 370/ESA, el tamaño de página es de 4 Kbytes y el tamaño de segmentos de 1 Mbyte. ¿Qué beneficios de la segmentación no se encuentran disponibles en este esquema? ¿Cuál es el beneficio del modelo segmentación de los 370?

8.10. Asumiendo un tamaño página de 4 Kbytes y que una entrada de la tabla páginas requiere 4 bytes, ¿Cuántos niveles de tablas de páginas se necesitarán para proyectar un espacio de direcciones de 64 bits si la tabla de páginas principal entra en una única página?

8.11. Considere un sistema con gestión de memoria basada en páginas y que utiliza un único nivel de páginas. Asumiendo que la tabla de páginas necesaria se encuentra siempre en la memoria.

a) Si una referencia a la memoria consume 200 ns, ¿cuánto puede tardar una referencia a una página de memoria?

b) Ahora añadimos una MMU que añade una sobrecarga de 20 ns, tanto si hay acierto como si no. Si se asume que el 85% de todas las referencias a memoria son aciertos en la TLB de la MMU, ¿cuál es el tiempo efectivo de acceso a la memoria?

c) Explíquese cómo la tasa de aciertos en la TLB afecta al tiempo de acceso a la memoria efectivo.

8.12. Considere una cadena de referencias a páginas de un proceso con un conjunto de trabajo de M marcos, inicialmente vacíos. La cadena de referencias a páginas es una longitud P con N números de páginas diferentes. Para cualquier algoritmo de reemplazo de páginas,

a) ¿Cuál es el límite inferior en el número de fallos de página?

b) ¿Cuál es el límite superior en el número de fallos de página?

8.13. En explicaciones sobre los diferentes algoritmos de reemplazo de páginas, un autor realiza la analogía con un quita-nieves moviéndose a lo largo de una pista circular. La nieve cae de forma uniforme a lo largo de toda la pista y el único quita-nieves se mueve a una velocidad constante a lo largo de todo el recorrido. La nieve una vez que ha pasado el quita-nieves por la pista desaparece del sistema.

a) ¿A cuál de los algoritmo de reemplazo de páginas presentados en la Sección 8.2 se puede asociar esta analogía?

b) ¿Qué sugiere esta analogía sobre el comportamiento del algoritmo de reemplazo de páginas en cuestión?

8.14. En la arquitectura S/370, una clave de almacenamiento es un campo de control asociado a cada marco de página de la memoria real. Dos de los bits de esa clave son importantes para el reemplazo de páginas y son el bit de referencia y el bit de cambio. El bit de referencia se pone a 1 cuando se ha accedido a una dirección dentro del marco para lectura o para escritura y se pone a 0 cuando se carga una nueva página en el marco. El bit de cambio se pone a 1 cuando se realiza una operación escritura sobre cualquier posición dentro del marco. Sugiera una estrategia para determinar cuál de los marcos de página se ha utilizado hace más tiempo, usando únicamente el bit de referencia.

8.15. Considere la siguiente secuencia de referencias a páginas (cada elemento de la secuencia representa un número de página):

1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5

Se refiere el *tamaño del conjunto de trabajo medio* después de la referencia *k*-ésima como

$s_k(\Delta) = \sum_{t=1}^{k} W(t,\Delta)$ y se define la *probabilidad de página ausente* como

$m_k(\Delta) = \frac{1}{k}\sum_{t=1}^{k} F(t,\Delta),$ donde F(t, Δ)=1 si ocurre un fallo de página en el tiempo virtual *t*, y 0 en otro caso.

a) Dibuje un diagrama similar al mostrado la Figura 8.19 para secuencia referencias, restringiéndose a los valores de Δ = 1, 2, 3, 4, 5, 6.

b) Dibuje s_{20} (Δ) en function de Δ.

c) Dibuje m_{20} (Δ) en function de Δ.

8.16. Una clave para el rendimiento de la política de gestión del conjunto reciente VSWS es el valor de *Q*. La experiencia ha demostrado que, para un valor fijo de *Q* en un determinado proceso, hay diferencias considerables en la secuencia de fallos de página a lo largo de las distintas etapas de la ejecución. Además, si se toma un único valor de *Q* para diferentes procesos, las frecuencias de fallos de página son considerablemente diferentes. Estas diferencias indican claramente que un mecanismo que ajuste dinámicamente el valor de *Q* durante el tiempo de vida de un proceso incrementaría considerablemente el rendimiento del algoritmo. Sugiera un mecanismo sencillo para este propósito.

8.17. Considere que una tarea se encuentra dividida en cuatro segmentos de igual tamaño y que el sistema construye una tabla de descriptores de páginas de ocho entradas por cada segmento. De esta forma, el sistema implementa una combinación de segmentación y paginación. Se asume que el tamaño de la página es de 2 Kbytes.

a) ¿Cuál es el tamaño máximo de los segmentos?

b) ¿Cuál es el tamaño máximo del espacio de direcciones lógicas de una tarea?

c) Supongamos que un elemento de la ubicación física 00021ABC va a ser accedido por la tarea, ¿cuál sería el formato de la dirección lógica que la tarea generaría para él? ¿Cuál es el espacio de direcciones físicas máximo para el sistema?

8.18. Considere un espacio de direcciones lógicas paginadas (compuesto por 32 páginas de 2 Kbytes cada una) que se proyecta en un espacio de memoria física de 1 Mbyte.

a) ¿Cuál es el formato de las direcciones lógicas del procesador?

b) ¿Cuál es la longitud y anchura de la tabla de páginas (no tenga en cuenta los bits de «derechos de acceso»)?

c) ¿Cuál sería el efecto en la tabla de páginas si el espacio de la memoria física sólo fuese la mitad?

8.19. El núcleo de UNIX proporciona crecimiento dinámico de la pila de proceso en la memoria virtual, pero nunca intentará reducirla. Considere el caso de un programa que llama a una subrutina escrita en C que pide que se reserve un vector local en la pila que consume 10 Kbytes. El núcleo expandirá el segmento de pila para poder ubicarlo. Cuando se regresa de la subrutina, el puntero de pila se ajusta y dicho espacio podría liberarse por parte del núcleo, pero no lo hace. Explique por qué sería posible reducir la pila en ese instante y por qué el núcleo de UNIX no lo hace.

APÉNDICE 8A TABLAS HASH

Consideremos el siguiente problema: un conjunto de *N* objetos que se van a almacenar en una tabla. Cada objeto consiste en una etiqueta y la información adicional, a la que nos referiremos como valor del objeto. Adicionalmente, sería deseable disponer de una serie de operaciones ordinarias sobre esta tabla, inserción, borrado y búsqueda de un valor por una determinada clave.

Si las etiquetas de los objetos son numéricas, en el rango de *0* a *M – 1*, la solución sencilla sería utilizar la tabla de longitud *M*. Un objeto con etiqueta *i* se insertaría en la tabla en la posición *i*, siempre y cuando los objetos sean de tamaño fijo, la búsqueda en la tabla es trivial e implica la indexación de la misma en base a la etiqueta numérica del objeto. Además, no sería necesario almacenar dicha etiqueta en la tabla, debido a que la propia posición del objeto implica su valor. Estas tablas se denominan **tablas de acceso directo**.

Si las etiquetas no son numéricas, aún así existe la posibilidad de utilizar la estrategia de acceso directo. Si nos referimos a los objetos como *A[1] ... A[N]*, cada objeto *A[i]* consiste en una etiqueta, k_i, y una clave, v_i. Si se define una función de proyección *I(k)*, de forma que *I(k)* tome valores entre *1* y *M* para todas las claves y que $I(k_i) \neq (k_j)$ para cualquier par de valores *i* y *j* diferentes. En este caso, también se pueden utilizar tablas de acceso directo, con una tabla de longitud *M*.

La principal dificultad de estos esquemas ocurre cuando *M* es mucho más grande que *N*. En este caso, la proporción de entradas sin usar en la tabla es muy grande, y esto implica un uso poco eficiente de la memoria. La alternativa sería la utilización de una tabla de longitud *M* para almacenar esos objetos (etiqueta y valor) en cada una de las entradas. En este esquema, la cantidad de memoria se minimiza pero hace muchísimo más pesado el proceso de consulta de la tabla. Las diferentes posibilidades son:

- **Búsqueda secuencial.** Esta es una estrategia de fuerza bruta que consume mucho tiempo para tablas de gran tamaño.
- **Búsqueda asociativa.** Con el soporte apropiado del hardware, todos los elementos de la tabla se pueden buscar de forma simultánea. Esa estrategia no es de propósito general y no se puede aplicar a todos los tipos de tablas.
- **Búsqueda binaria.** Si las etiquetas o las proyecciones numérica de las etiquetas se ordenan de forma ascendente en la tabla, la búsqueda binaria es mucho más rápida que la secuencia (Tabla 8.6) y no requiere hardware especial.

La búsqueda binaria parece prometedora para la consulta de estas tablas. La principal desventaja de este método es que la inserción del objeto no es un proceso simple y habitualmente requerirá la reordenación de todas las entradas. Por tanto, la búsqueda binaria se usa habitualmente en tablas que son razonablemente estáticas y que sufren pocos cambios.

TABLA 8.6. Tiempo de búsqueda medio para uno de los *N* objetos de una tabla de tamaño *M*.

Técnica	**Longitud de la búsqueda**
Directa	1
Secuencial	$\frac{M+1}{2}$
Binaria	$\log_2(M)$
Hashing lineal	$\frac{2 - N/M}{2 - 2N/M}$
Hash (desbordamiento encadenado)	$1+\frac{N-1}{2M}$

Queremos evitar las penalizaciones de memoria de una estrategia de acceso directo sencillo y las penalizaciones de proceso de las alternativas vistas anteriormente. El método utilizado más habitualmente para conseguir esto se denomina ***hashing***. El *hashing*, que fue desarrollado en los años 50, es muy sencillo de implementar y tiene dos ventajas. En primer lugar, puede consultar la mayoría de objetos en un acceso directo, como el caso que hemos visto antes, en segundo lugar, las inserciones y los borrados se pueden manejar sin ninguna complejidad adicional.

La función de *hashing* se puede definir de la siguiente manera. Si asumimos que tenemos *N* objetos que van a ser almacenados en una **tabla *hash*** de longitud *M*, con $M \geq N$, pero no mucho mayor. Para insertar un objeto en la tabla,

I1. Se convierte la etiqueta del objeto en un valor pseudo-aleatorio *n* que se encuentra entre 0 y $M - 1$. Por ejemplo, una función de proyección habitual es dividir la etiqueta entre *M* y quedarse con el resto del valor como *n*.

I2. Se usa *n* como índice en la tabla *hash*.

a) Su entrada correspondiente a la tabla esta vacía, almacenar el objeto (etiqueta y valor) en dicha entrada.

b) Si la entrada está ocupada, el almacenamiento del objeto ha causado una colisión, y se almacenará el objeto en un área de desbordamiento, como veremos más adelante.

Para la operación de consulta del objeto del cual conocemos la etiqueta,

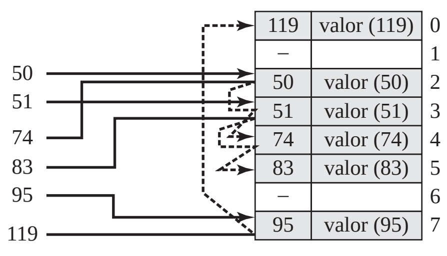
C1. Se convierte la etiqueta del objeto en un valor pseudo-aleatorio que se encuentra entre 0 y $M - 1$, usando la misma función de proyección que en el caso de la inserción.

C2. Se usa *n* como índice en la tabla *hash*.

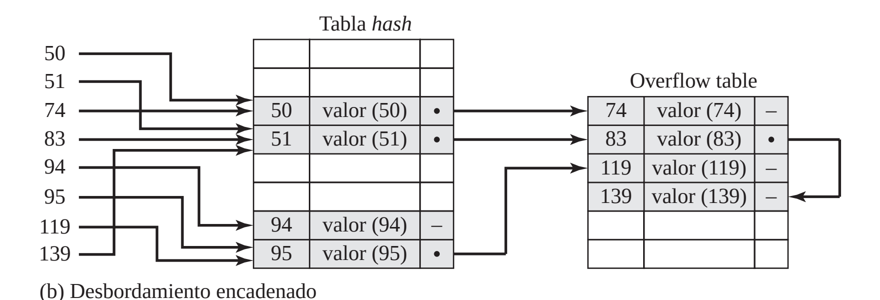
a) Si la entrada correspondiente en la tabla se encuentra vacía, entonces el objeto no se encontraba almacenado previamente en la tabla.

b) Si la entrada se encuentra ocupada y las etiquetas coinciden, entonces se puede recuperar el valor.

c) Si la entrada se encuentra ocupada y las etiquetas no coinciden, entonces se continuará la consulta en el área de desbordamiento.



(a) *Hashing* lineal



(b) Desbordamiento encadenado

Figura 8.27. *Hashing.*

Los esquemas de *hashing* difieren en la forma la en que se manejan las colisiones y los desbordamientos. Una técnica habitual se denomina ***hashing* lineal** y se usa habitualmente en compiladores. En esta técnica, la regla I2.b se convierte en

I2.b Si la entrada está ocupada, $n = n+1$ (mod M) y se vuelve al paso I2.a.

La regla C2.c se modifica consecuentemente.

La Figura 8.27a es un ejemplo. En este caso, las etiquetas de los objetos que se almacenan son numéricas, y la tabla *hash* tiene ocho posiciones ($M=8$). La función de proyección consiste en quedarse con el resto de la división entre 8. La figura que asume los objetos fueron insertados en orden ascendente, sin que esto sea necesario. De esta forma, los objetos 50 y 51 se proyectan en las posiciones 2 y 3, respectivamente, como están vacías se insertan en ellas. El objeto 74 también se proyecta en la posición 2, pero ésta no se encuentra vacía, se intentará en la posición 3. Como ésta está también ocupada, se intentará en la posición 4, la cual está libre.

No es fácil determinar la longitud media de la operación de consulta para un determinado objeto en una tabla así debido a este efecto de agrupamiento. Una fórmula aproximada fue obtenida por Schay y Spruth [SCHA62]:

$$\text{Longitud media de búsqueda} \quad \frac{2-r}{2-2r}$$

donde $r = N/M$. Nótese que el resultado es independiente del tamaño de la tabla y depende sólo de cómo se encuentre de llena. El resultado sorprendente es que para una tabla que se encuentra al 80% de ocupación, la longitud media de una búsqueda se encuentra en torno a 3.

Incluso así, una búsqueda de longitud 3 se puede considerar larga, y las tablas de *hashing* lineal tienen la desventaja adicional que no resulta fácil borrar elementos de ellas. Una estrategia alter-

nativa, que proporciona longitud de búsqueda más corta (Tabla 8.6) y permite borrados de la misma forma que las inserciones, es el **desbordamiento encadenado**. Esta técnica se muestra en la Figura 8.27b. En este caso, hay una tabla separada a la cual se redirigen las operaciones de colisión. Esta tabla incluye punteros entre las entradas asociándolas a una cadena con cualquier posición de la tabla *hash*. En este caso la longitud media de búsqueda, asumiendo datos distribuidos de forma aleatoria, es:

$$\text{Longitud media de búsqueda} = 1 + \frac{N-1}{2M}$$

Para valores grandes N y M, este valor se aproxima a 1,5 para el caso de $M = N$. De esta forma, esta técnica proporciona un almacenamiento compacto con una operación de consulta rápida.

PARTE IV

PLANIFICACIÓN

Un sistema operativo debe asignar los recursos del computador entre las necesidades potencialmente competitivas de múltiples procesos. En el caso del procesador, el recurso que se debe asignar es el tiempo de ejecución en el procesador. La forma de asignarlo es la planificación. La función de planificación debe estar diseñada para satisfacer varios objetivos que incluyen equidad, ausencia de inanición de cualquier proceso, uso eficiente del tiempo del procesador y poca sobrecarga. Además, la función de planificación puede necesitar tener en cuenta diferentes niveles de prioridad o plazos de tiempo real para el inicio o finalización de ciertos procesos. Durante años, la planificación ha sido objeto de profundas investigaciones, y se han implementado múltiples algoritmos. Hoy en día, la investigación sobre la planificación se centra en la explotación de sistemas multiprocesador, particularmente para aplicaciones multihilo, y en la planificación de tiempo real.

GUÍA DE LECTURA DE LA PARTE CUATRO

CAPÍTULO 9. PLANIFICACIÓN UNIPROCESADOR

El Capítulo 9 se centra en la planificación con un solo procesador. En este contexto limitado se pueden definir y clarificar múltiples aspectos de diseño relacionados con la planificación. El Capítulo 9 comienza con un examen de los tres tipos de planificación del procesador: a largo plazo, a medio plazo y a corto plazo. El grueso del capítulo está dedicado a los temas de la planificación a corto plazo. Se examinan y comparan varios algoritmos propuestos.

CAPÍTULO 10. PLANIFICACIÓN MULTIPROCESADOR Y DE TIEMPO REAL

El Capítulo 10 examina dos áreas centrales en la investigación actual sobre la planificación. La presencia de múltiples procesadores complica las decisiones de la planificación, pero abre nuevas posibilidades. En particular, con múltiples procesadores se puede planificar la ejecución simultánea de múltiples hilos del mismo proceso. La primera parte del Capítulo 10 pasa revista a la planificación multiprocesador y multihilo. El resto del capítulo trata de la planificación de tiempo real. Los requisitos de tiempo real son los que más exigen de un planificador, porque van más allá de la imparcialidad o de la prioridad, al especificar límites temporales para el comienzo o finalización de diversas tareas o procesos.

CAPÍTULO 9

Planificación uniprocesador

◆ ◆

En un sistema multiprogramado, hay múltiples procesos de modo concurrente en la memoria principal. Los procesos pueden estar usando un procesador o pueden estar esperando el suceso de algún evento, tal como la finalización de una operación de E/S. El procesador o los procesadores se mantienen ocupados ejecutando un proceso mientras el resto espera.

La clave de la multiprogramación es la planificación. De hecho, se diferencian cuatro tipos de planificación (Tabla 9.1). Una de ellas, la planificación de la E/S, se trata más a fondo en el Capítulo 11, donde se habla de la E/S. Los tres tipos restantes de planificación, pertenecientes a la planificación del procesador, se tratan en este capítulo y en el siguiente.

Este capítulo comienza examinando los tres tipos de planificación del procesador, mostrando cómo se relacionan. Se verá cómo la planificación a largo plazo y la planificación a medio plazo se preocupan por aspectos de rendimiento relacionados con el grado de multiprogramación. Estos temas se tratan en el Capítulo 3 y en más detalle en los Capítulos 7 y 8. De esta forma, la parte restante de este capítulo se centra en la planificación a corto plazo y se limita a la planificación en un sistema uniprocesador. Ya que el uso de múltiples procesadores añade complejidad adicional, es mejor centrarse primero en el caso de uniprocesadores, de forma que las diferencias entre los algoritmos de planificación se pueden ver con mayor claridad.

La Sección 9.2 analiza varios algoritmos que se pueden utilizar para tomar decisiones sobre la planificación a corto plazo.



9.1. TIPOS DE PLANIFICACIÓN DEL PROCESADOR

El objetivo de la planificación de procesos es asignar procesos a ser ejecutados por el procesador o procesadores a lo largo del tiempo, de forma que se cumplan los objetivos del sistema tales como el tiempo de respuesta, el rendimiento y la eficiencia del procesador. En muchos sistemas, esta actividad de planificación se divide en tres funciones independientes: planificación a largo-, medio-, y corto-plazo. Los nombres sugieren las escalas de tiempo relativas con que se ejecutan estas funciones.

Tabla 9.1. Tipos de planificación.

Planificación a largo plazo	La decisión de añadir un proceso al conjunto de procesos a ser ejecutados
Planificación a medio plazo	La decisión de añadir un proceso al número de procesos que están parcialmente o totalmente en la memoria principal
Planificación a corto plazo	La decisión por la que un proceso disponible será ejecutado por el procesador
Planificación de la E/S	La decisión por la que un proceso que está pendiente de una petición de E/S será atendido por un dispositivo de E/S disponible

La Figura 9.1 relaciona las funciones de planificación con el diagrama de transición de estados de los procesos (anteriormente visto en la Figura 3.9b). La planificación a largo plazo se realiza cuando se crea un nuevo proceso. Hay que decidir si se añade un nuevo proceso al conjunto de los que están activos actualmente. La planificación a medio plazo es parte de la función de intercambio (*swapping function*). Hay que decidir si se añade un proceso a los que están al menos parcialmente en memoria y que, por tanto, están disponibles para su ejecución. La planificación a corto plazo conlleva decidir cuál de los procesos listos para ejecutar es ejecutado. La Figura 9.2 reorganiza el

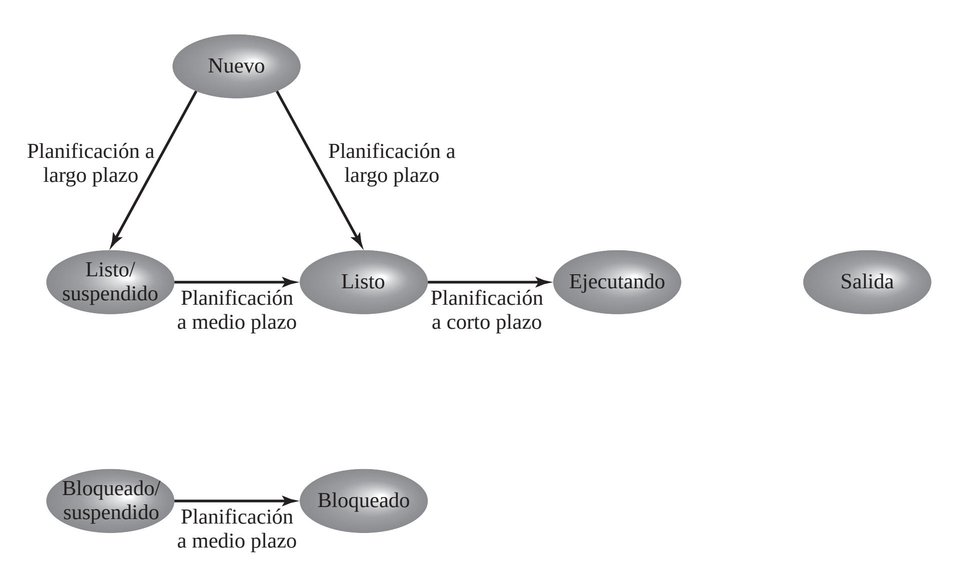


Figura 9.1. Planificación y transiciones de estado de los procesos.

diagrama de transición de estados de la Figura 3.9b para que se pueda apreciar el anidamiento de las funciones de planificación.

La planificación afecta al rendimiento del sistema porque determina qué proceso esperará y qué proceso progresará. Este punto de vista se presenta en la Figura 9.3, donde se muestran las colas involucradas en los estados de transición de los procesos[1]. Fundamentalmente, la planificación es un problema de manejo de colas para minimizar el retardo en la cola y para optimizar el rendimiento en un entorno de colas.

PLANIFICACIÓN A LARGO PLAZO

El planificador a largo plazo determina qué programas se admiten en el sistema para su procesamiento. De esta forma, se controla el grado de multiprogramación. Una vez admitido, un trabajo o programa de usuario se convierte en un proceso y se añade a la cola del planificador a corto plazo. En algunos sistemas, un proceso de reciente creación comienza en la zona de intercambio, en cuyo caso se añaden a la cola del planificador a medio plazo.

En un sistema por lotes, o en la parte de lotes de un sistema operativo de propósito general, los nuevos trabajos enviados se mandan al disco y se mantienen en una cola de lotes. El planificador a largo plazo creará procesos desde la cola siempre que pueda. En este caso hay que tomar dos decisiones. La primera, el planificador debe decidir cuándo el sistema operativo puede coger uno o más procesos adicionales. La segunda, el planificador debe decidir qué trabajo o trabajos se aceptan y son convertidos en procesos. Consideremos brevemente estas dos decisiones.

La decisión de cuándo crear un nuevo proceso se toma dependiendo del grado de multiprogramación deseado. Cuanto mayor sea el número de procesos creados, menor será el porcentaje de tiempo

---

[1] Por simplicidad, la Figura 9.3 muestra a los nuevos procesos yendo directamente al estado Listo, mientras que las Figuras 9.1 y 9.2 muestran dos opciones, al estado de Listo o al estado de Listo/Suspendido.

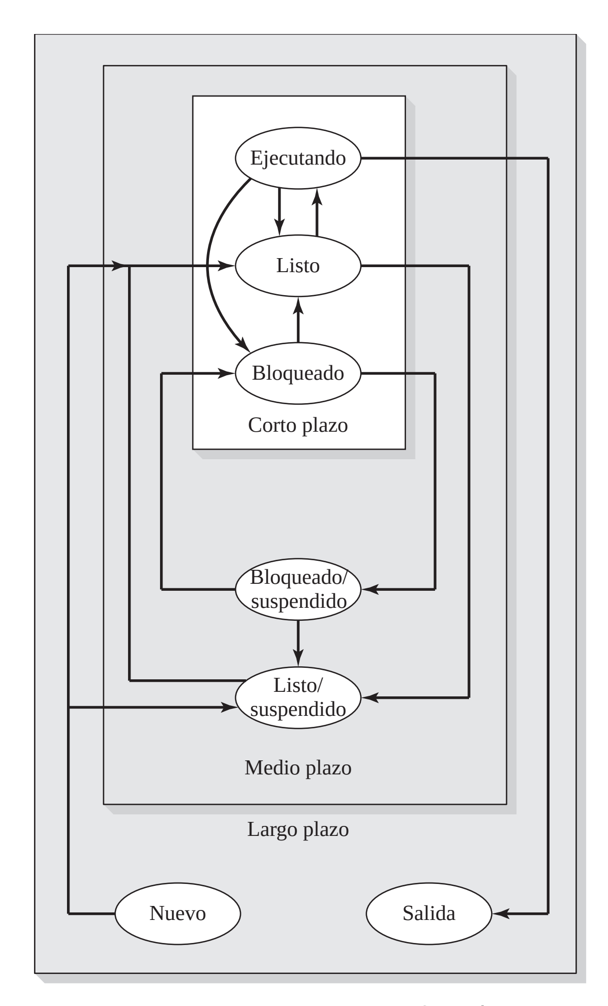


Figura 9.2. Niveles de planificación.

en que cada proceso se pueda ejecutar (es decir, más procesos compiten por la misma cantidad de tiempo de procesador). De esta forma, el planificador a largo plazo puede limitar el grado de multiprogramación a fin de proporcionar un servicio satisfactorio al actual conjunto de procesos. Cada vez que termine un trabajo, el planificador puede decidir añadir uno o más nuevos trabajos. Además, si la fracción de tiempo que el procesador está ocioso excede un determinado valor, se puede invocar al planificador a largo plazo.

La decisión de qué trabajo admitir el siguiente, puede basarse en un sencillo «primero en llegar primero en servirse», o puede ser una herramienta para gestionar el rendimiento del sistema. El criterio utilizado puede incluir la prioridad, el tiempo estimado de ejecución y los requisitos de E/S. Por ejemplo, si la información está disponible, el planificador puede intentar encontrar un compromiso entre procesos limitados por el procesador y procesos limitados por la E/S[2]. Además, la decisión pue-

[2] Se dice que un proceso está limitado por el procesador si realiza mucho trabajo computacional y ocasionalmente usa los dispositivos de E/S. Se dice que un proceso está limitado por la E/S si se pasa más tiempo utilizando los dispositivos de E/S que el procesador.

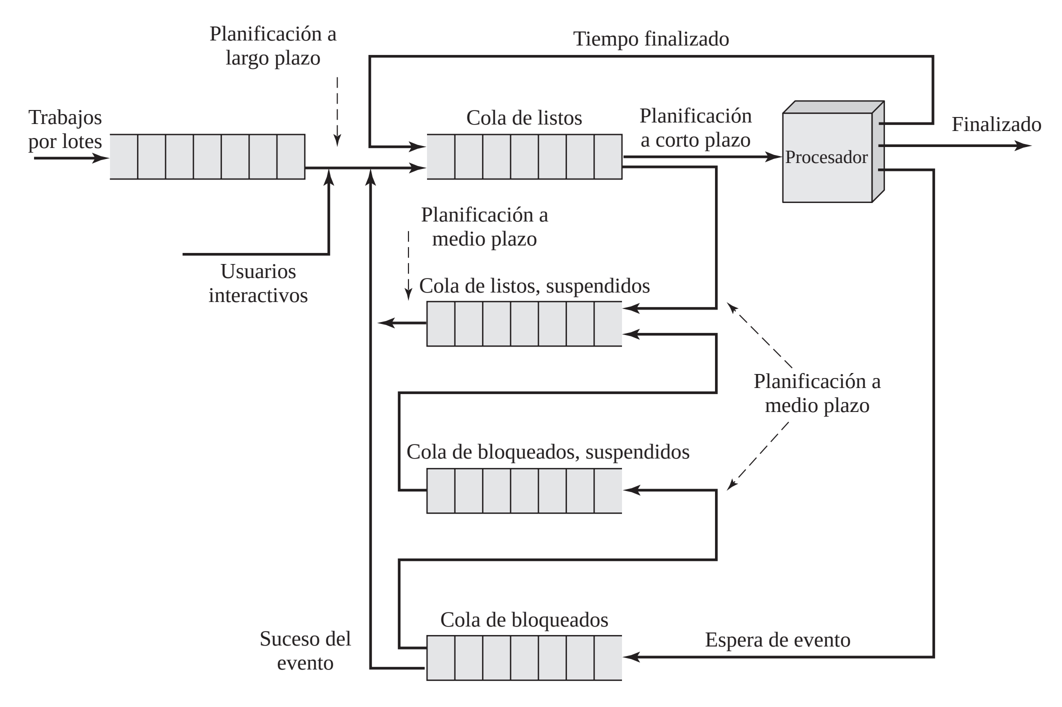


Figura 9.3. Diagrama de encolamiento para la planificación.

de ser tomada dependiendo de los recursos de E/S que vayan a ser utilizados, de forma que se intente equilibrar el uso de la E/S.

Para los programas interactivos en un sistema de tiempo compartido, la petición de la creación de un proceso, puede estar generada por un usuario intentando conectarse al sistema. Los usuarios de tiempo compartido no se sitúan simplemente en una cola hasta que el sistema los pueda aceptar. Más exactamente, el sistema operativo aceptará a todos los usuarios autorizados hasta que el sistema se sature La saturación se estima utilizando ciertas medidas prefijadas del sistema. Llegado a este punto, una petición de conexión se encontrará con un mensaje indicando que el sistema está completo y el usuario debería reintentarlo más tarde.

PLANIFICACIÓN A MEDIO PLAZO

La planificación a medio plazo es parte de la función de intercambio. Los aspectos relacionados se discuten en los Capítulos 3, 7 y 8. Con frecuencia, la decisión de intercambio se basa en la necesidad de gestionar el grado de multiprogramación. En un sistema que no utiliza la memoria virtual, la gestión de la memoria es también otro aspecto a tener en cuenta. De esta forma, la decisión de meter un proceso en la memoria, tendrá en cuenta las necesidades de memoria de los procesos que están fuera de la misma.

PLANIFICACIÓN A CORTO PLAZO

En términos de frecuencia de ejecución, el planificador a largo plazo ejecuta con relativamente poca frecuencia y toma la decisión de grano grueso de admitir o no un nuevo proceso y qué proceso admi-

tir. El planificador a medio plazo se ejecuta más frecuentemente para tomar decisiones de intercambio. El planificador a corto plazo, conocido también como activador, ejecuta mucho más frecuentemente y toma las decisiones de grano fino sobre qué proceso ejecutar el siguiente.

El planificador a corto plazo se invoca siempre que ocurre un evento que puede conllevar el bloqueo del proceso actual y que puede proporcionar la oportunidad de expulsar al proceso actualmente en ejecución en favor de otro. Algunos ejemplos de estos eventos son

- Interrupciones de reloj.
- Interrupciones de E/S.
- Llamadas al sistema.
- Señales (por ejemplo, semáforos).

9.2. ALGORITMOS DE PLANIFICACIÓN

CRITERIOS DE LA PLANIFICACIÓN A CORTO PLAZO

El objetivo principal de la planificación a corto plazo es asignar tiempo de procesador de tal forma que se optimicen uno o más aspectos del comportamiento del sistema. Generalmente, se establece un conjunto de criterios con los que se pueden evaluar varias políticas de planificación.

Los criterios utilizados habitualmente se pueden categorizar en dos dimensiones. Primero, se puede hacer una distinción entre criterios orientados al usuario y criterios orientados al sistema. Los criterios orientados al usuario están relacionados con el comportamiento del sistema tal y como lo percibe un usuario individual o un proceso. Un ejemplo es el tiempo de respuesta en un sistema interactivo. El tiempo de respuesta es el tiempo que transcurre entre el envío de una petición y la aparición de la respuesta. Esta cantidad es visible por parte del usuario y lógicamente es de su interés. Nos gustaría tener una política de planificación que proporcione «buen» servicio a varios usuarios. En el caso del tiempo de respuesta se puede definir un límite, por ejemplo, 2 segundos. De esta forma el objetivo del mecanismo de planificación puede ser maximizar el número de usuarios que tienen un tiempo de respuesta medio menor o igual a 2 segundos.

Otros criterios son los relacionados con el sistema. Es decir, la atención se centra en el uso efectivo y eficiente del procesador. Un ejemplo es el rendimiento (*throughput*), que es la tasa con que los procesos se finalizan. Esta medida es valiosa en relación al sistema y es algo que nos gustaría maximizar. Sin embargo, no se centra en los servicios proporcionados al usuario. De esta forma, el rendimiento incumbe al administrador del sistema pero no a los usuarios.

Mientras que los criterios orientados al usuario son importantes en prácticamente todos los sistemas, los criterios orientados al sistema son generalmente menos importantes en sistemas de un solo usuario. En sistemas de un solo usuario, probablemente no es tan importante lograr una alta utilización del procesador o un alto rendimiento como que la respuesta del sistema a las aplicaciones del usuario sea aceptable.

También es posible clasificar los criterios dependiendo de si están o no relacionados con las prestaciones. Los criterios relacionados con las prestaciones son cuantitativos y generalmente pueden ser medidos. Algunos ejemplos son el tiempo de respuesta y el rendimiento. Los criterios que no están relacionados con las prestaciones, o son cualitativos por naturaleza, o no pueden ser medidos y analizados. Un ejemplo de tales criterios es la previsibilidad. Nos gustaría que el servicio proporcionado a los usuarios tuviera las mismas características a lo largo del tiempo, independientemente de otros trabajos que se estén realizando en el sistema. Hasta cierto punto este criterio pue-

de ser medido calculando las discrepancias en función de la carga de trabajo. Sin embargo, esta medida no es tan directa como medir el rendimiento o el tiempo de respuesta como una función de la carga de trabajo.

La Tabla 9.2 resume los criterios clave de la planificación. Son independientes, y es imposible optimizar todos ellos de forma simultánea. Por ejemplo, proporcionar un buen tiempo de respuesta puede requerir un algoritmo de planificación que cambie entre procesos frecuentemente. Esto incrementa la sobrecarga del sistema, reduciendo las prestaciones. De esta forma, el diseño de las políticas de un planificador implica un compromiso entre requisitos competitivos; los pesos relativos dados a los distintos requisitos dependerán de la naturaleza y el uso dado al sistema.

Tabla 9.2. Criterios de planificación.

Orientados al usuario, relacionados con las prestaciones

Tiempo de estancia (*turnaround time*) Tiempo transcurrido desde que se lanza un proceso hasta que finaliza. Incluye el tiempo de ejecución sumado con el tiempo de espera por los recursos, incluyendo el procesador. Es una medida apropiada para trabajos por lotes.

Tiempo de respuesta (*response time*) Para un proceso interactivo, es el tiempo que trascurre desde que se lanza una petición hasta que se comienza a recibir la respuesta. A menudo un proceso puede producir alguna salida al usuario mientras continúa el proceso de la petición. De esta forma, desde el punto de vista del usuario, es una medida mejor que el tiempo de estancia. La planificación debe intentar lograr bajos tiempos de respuesta y maximizar el número de usuarios interactivos con tiempos de respuesta aceptables.

Fecha tope (*deadlines*) Cuando se puede especificar la fecha tope de un proceso, el planificador debe subordinar otros objetivos al de maximizar el porcentaje de fechas tope conseguidas.

Orientados al usuario, otros

Previsibilidad Un trabajo dado debería ejecutarse aproximadamente en el mismo tiempo y con el mismo coste a pesar de la carga del sistema. Una gran variación en el tiempo de respuesta o en el tiempo de estancia es malo desde el punto de vista de los usuarios. Puede significar una gran oscilación en la sobrecarga del sistema o la necesidad de poner a punto el sistema para eliminar las inestabilidades.

Orientados al sistema, relacionados con las prestaciones

Rendimiento La política de planificación debería intentar maximizar el número de procesos completados por unidad de tiempo. Es una medida de cuánto trabajo está siendo realizado. Esta medida depende claramente de la longitud media de los procesos, pero está influenciada por la política de planificación, que puede afectar a la utilización.

Utilización del procesador Es el porcentaje de tiempo que el procesador está ocupado. Para un sistema compartido costoso, es un criterio significativo. En un sistema de un solo usuario y en otros sistemas, tales como los sistemas de tiempo real, este criterio es menos importante que algunos otros.

Orientados al sistema, otros

Equidad En ausencia de orientación de los usuarios o de orientación proporcionada por otro sistema, los procesos deben ser tratados de la misma manera, y ningún proceso debe sufrir inanición.

Imposición de prioridades Cuando se asignan prioridades a los procesos, la política del planificador debería favorecer a los procesos con prioridades más altas.

Equilibrado de recursos La política del planificador debería mantener ocupados los recursos del sistema. Los procesos que utilicen poco los recursos que en un determinado momento están sobreutilizados, deberían ser favorecidos. Este criterio también implica planificación a medio plazo y a largo plazo.

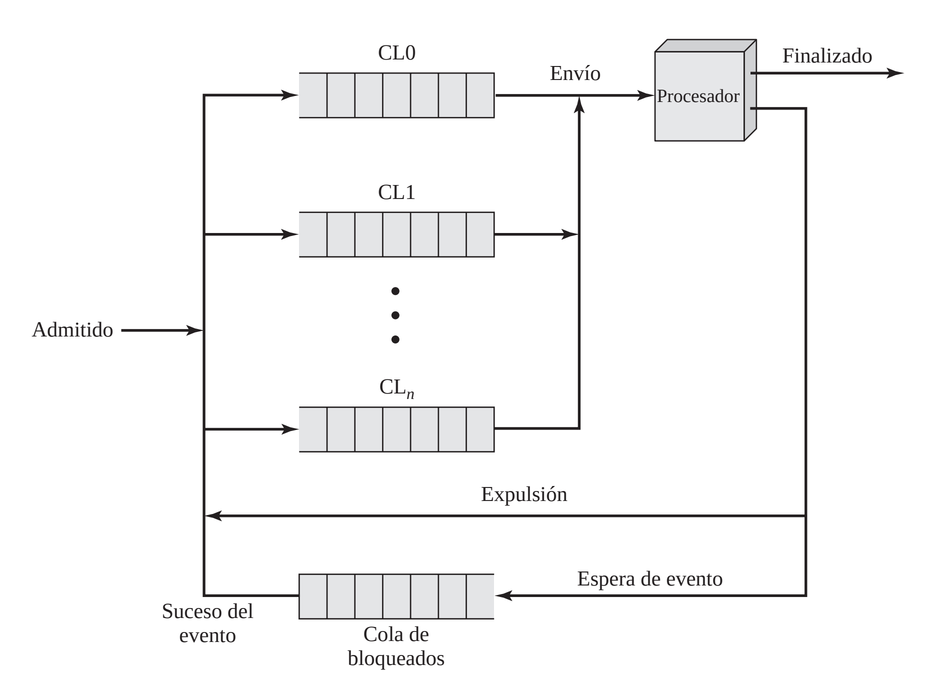


Figura 9.4. Encolamiento con prioridades.

En la mayor parte de los sistemas operativos interactivos, ya sean de un solo usuario o de tiempo compartido, un adecuado tiempo de respuesta es un requisito crítico. Debido a la importancia de este requisito, y ya que la definición de adecuación varía entre las aplicaciones, este tema se explora más en profundidad en un apéndice de este libro.

EL USO DE PRIORIDADES

En muchos sistemas, a cada proceso se le asigna una prioridad y el planificador siempre elegirá un proceso de prioridad mayor sobre un proceso de prioridad menor. La Figura 9.4 muestra el uso de las prioridades. Por claridad, el diagrama de colas está simplificado, ignorando la existencia de múltiples colas bloqueadas y de estados suspendidos (comparar con la Figura 3.8a). En lugar de una sola cola de procesos listos para ejecutar, se proporcionan un conjunto de colas en orden descendente de prioridad: CL0, CL1, ... CL*n*, con la prioridad[CL*i*]>prioridad[CL*j*] para $i<j$[3]. Cuando se va a realizar una selección en la planificación, el planificador comenzará en la cola de listos con la prioridad más alta (CL0). Si hay uno o más procesos en la cola, se selecciona un proceso utilizando alguna política de planificación. Si CL0 está vacía, entonces se examina CL1, y así sucesivamente.

Un problema de los esquemas de planificación con prioridades es que los procesos con prioridad más baja pueden sufrir inanición. Esto sucederá si hay siempre un conjunto de procesos de mayor prioridad listos para ejecutar. Si este comportamiento no es deseable, la prioridad de un proceso pue-

---

[3] En UNIX y otros muchos sistemas, los valores mayores de prioridad representan procesos de prioridad más baja; a menos que se esfecifique, seguiremos esta convención. Algunos sistemas, tales como Windows, utilizan la convención opuesta: un número mayor significa una mayor prioridad.

de cambiar con su antigüedad o histórico de ejecución. Posteriormente daremos un ejemplo de esta situación.

POLÍTICAS DE PLANIFICACIÓN ALTERNATIVAS

La Tabla 9.3 presenta información resumida sobre las políticas de planificación examinadas en esta subsección. La **función de selección** determina qué proceso, entre los procesos listos, se selecciona para su ejecución. La función puede estar basada en prioridades, requisitos sobre los recursos, o las características de ejecución del proceso. En el último caso, hay tres medidas cuantitativas:

w = tiempo usado en el sistema hasta este momento, esperando o ejecutando

e = tiempo usado en ejecución hasta este momento

s = tiempo total de servicio requerido por el proceso, incluyendo e; generalmente, esta cantidad debe ser estimada o proporcionada por el usuario

Por ejemplo, la función de selección max[w] implica una planificación del tipo *primero en llegar, primero en servirse* (first-come-first-served – FCFS).

El **modo de decisión** especifica los instantes de tiempo en que se ejecuta la función de selección. Hay dos categorías generales:

- **Sin expulsion (*nonpreemptive*).** En este caso, una vez que el proceso está en el estado Ejecutando, continúa ejecutando hasta que (a) termina o (b) se bloquea para esperar E/S o para solicitar algún servicio al sistema operativo.
- **Con expulsión (*preemptive*).** Un proceso ejecutando en un determinado momento puede ser interrumpido y pasado al estado de listo por el sistema operativo. La decisión de expulsar puede ser tomada cuando llega un nuevo proceso, cuando llega una interrupción que pasa un proceso de bloqueado a estado de listo, o periódicamente, basándose en las interrupciones del reloj.

Las políticas expulsivas tienen mayor sobrecarga que las no expulsivas, pero pueden proporcionar mejor servicio a la población total de procesos, ya que previenen que cualquier proceso pueda monopolizar el procesador durante mucho tiempo. Además, el coste de la expulsión puede resultar relativamente bajo a través de la utilización de mecanismos eficientes de cambios de proceso (tanta ayuda del hardware como sea posible) y proporcionando una gran cantidad de memoria principal para dejar un alto porcentaje de programas en la memoria principal.

A medida que se describan las políticas de planificación se utilizará el conjunto de procesos de la Tabla 9.4 como ejemplo de ejecución. Podemos pensar en estos procesos como trabajos por lotes, con un tiempo de servicio igual al tiempo de ejecución requerido. También los podemos considerar como procesos en curso que requieren un uso alternativo del procesador y de la E/S de forma repetitiva. En este último caso, los tiempos de servicio representan el tiempo de procesador requerido en un ciclo. En cualquier caso, en términos de un modelo de colas, esta cantidad se corresponde con el tiempo de servicio[4].

---

[4] El Apéndice 9B contiene un resumen de la terminología del modelo de colas.

Tabla 9.3. Características de algunas políticas de planificación.

	Función de Selección	Modo de Decisión	Rendimiento	Tiempo de Respuesta	Rendimiento	Efecto sobre los Procesos	Inanición
FCFS	max[*w*]	No expulsiva	No especificado	Puede ser alto especialmente si hay mucha diferencia entre los tiempos de ejecución de los procesos	Mínima	Penaliza procesos cortos; penaliza procesos con mucha E/S	No
Turno Rotatorio *(round robin)*	constante	Expulsiva (por rodajas de tiempo)	Puede ser mucho si la rodaja es demasiado pequeña	Proporciona buen tiempo de respuesta para procesos cortos	Mínima	Tratamiento justo	No
SPN	min[*s*]	No expulsiva	Alto	Proporciona buen tiempo de respuesta para procesos cortos	Puede ser alta	Penaliza procesos largos	Posible
SRT	min[*s*-*e*]	Expulsiva (a la llegada)	Alto	Proporciona buen tiempo de respuesta	Puede ser alta	Penaliza procesos largos	Posible
HRRN	max(*w*+*s*/*s*)	No expulsiva	Alto	Proporciona buen tiempo de respuesta	Puede ser alta	Buen equilibrio	No
Feedback	(ver texto)	Expulsiva (por rodajas de tiempo)	No especificado	No especificado	Puede ser alta	Puede favorecer procesos con mucha E/S	Posible

w = tiempo de espera
e = tiempo de ejecución hasta el momento
s = tiempo total de servicio requerido por el proceso, incluyendo *e*

Para el ejemplo de la Tabla 9.4, la Figura 9.5 muestra los patrones de ejecución para cada directiva en cada ciclo, y la Tabla 9.5 resume los principales resultados. Primero, se determina el tiempo de finalización de cada proceso. De esta forma, se puede determinar el tiempo de estancia. En términos del modelo de colas, el **tiempo de estancia** (*turnaround time* – TAT) es el tiempo de residencia T_n o tiempo total que un elemento está en el sistema (tiempo de espera, más tiempo de servicio). Algo más útil es el tiempo de estancia normalizado, que es tasa del tiempo de estancia y tiempo de servicio. Este valor indica el retardo relativo experimentado por un proceso. Así, mayor sea el tiempo de ejecución, mayor será el valor absoluto del retardo que se puede tolerar. El menor valor posible de esta tasa es 1,0; valores mayores corresponden a un servicio de nivel menor.

Tabla 9.4. Ejemplo de planificación de procesos.

Proceso	Tiempo de llegada	Tiempo de servicio
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Primero en llegar, primero en servirse (*first-come-first-served*). La directiva de planificación más sencilla es primero en llegar primero en servirse (FCFS), también conocida como primero-entra-primero-sale (FIFO) o como un sistema de colas estricto. En el momento en que un proceso pasa al estado de listo, se une a la cola de listos. Cuando el proceso actualmente en ejecución deja de ejecutar, se selecciona para ejecutar el proceso que ha estado más tiempo en la cola de listos.

FCFS funciona mucho mejor para procesos largos que para procesos cortos. Considérese el siguiente ejemplo, basado en [FINK88]:

Proceso	Tiempo de Llegada	Tiempo de Servicio (T_s)	Tiempo de Comienzo	Tiempo de Finalización	Tiempo de Estancia (T_r)	T_r/T_s
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1,99
Media					100	26

El tiempo de estancia normalizado para el proceso Y es excesivamente grande en comparación con los otros procesos: el tiempo total que está en el sistema es 100 veces el tiempo requerido para su proceso. Esto sucederá siempre que llegue un proceso corto a continuación de un proceso largo. Por otra parte, incluso en este ejemplo extremo, los procesos largos no van mal. El proceso Z tiene un tiempo de estancia de casi el doble que Y, pero su tiempo de residencia normalizado está por debajo de 2,0.

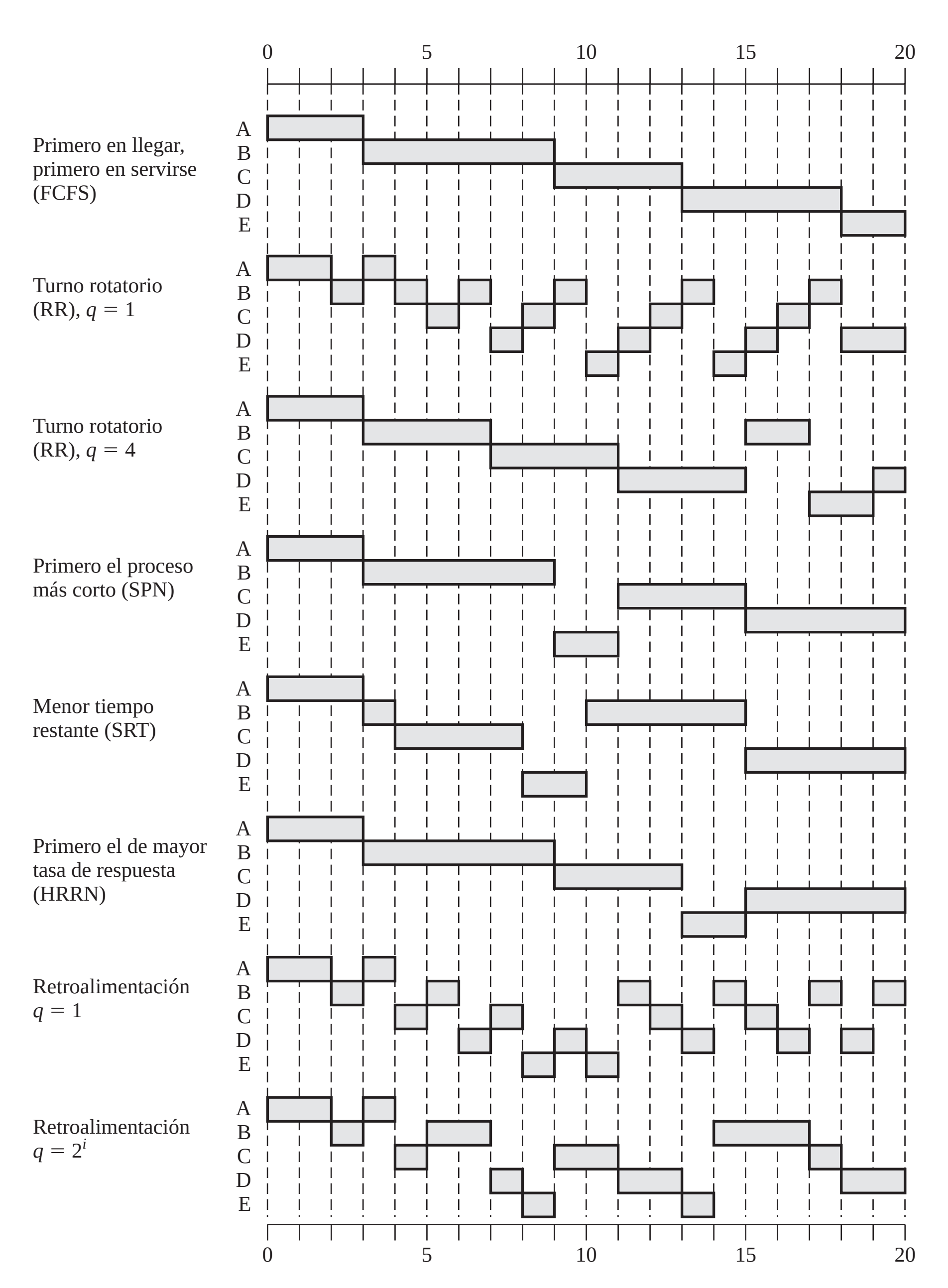


Figura 9.5. Comparación de políticas de planificación.

Otro problema de FCFS es que tiende a favorecer procesos limitados por el procesador sobre los procesos limitados por la E/S. Considere que hay una colección de procesos, uno de los cuales está limitado por el procesador y un número de procesos limitados por la E/S. Cuando el proceso limitado por el procesador está ejecutando, el resto de los procesos debe esperar. Alguno de estos estarán en las colas de E/S (estado bloqueado) pero se pueden mover a la cola de listos mientras que el proceso limitado por el procesador sigue ejecutando. En este punto, la mayor parte de los dispositivos de E/S pueden estar ociosos, incluso aunque exista trabajo potencial que pueden hacer. Cuando el proceso

actual en ejecución deja el estado Ejecutando, los procesos listos pasarán al estado de Ejecutando y se volverán a bloquear en un evento de E/S. Si el proceso limitado por el procesador está también bloqueado, el procesador se quedará ocioso. De esta manera, FCFS puede conllevar usos ineficientes del procesador y de los dispositivos de E/S.

FCFS no es una alternativa atractiva por sí misma para un sistema uniprocesador. Sin embargo, a menudo se combina con esquemas de prioridades para proporcionar una planificación eficaz. De esta forma, el planificador puede mantener varias colas, una por cada nivel de prioridad, y despachar dentro de cada cola usando primero en llegar primero en servirse. Veremos un ejemplo de este sistema más adelante, cuando hablemos de la planificación retroalimentada (*feedback*).

Turno rotatorio (*round robin*) Una forma directa de reducir el castigo que tienen los trabajos cortos con FCFS es la utilización de expulsión basándose en el reloj. La política más sencilla es la del turno rotatorio, también denominada planificación cíclica. Se genera una interrupción de reloj cada cierto intervalo de tiempo. Cuando sucede la interrupción, el proceso actual en ejecución se sitúa en la cola de listos, y se selecciona el siguiente trabajo según la política FCFS. Esta técnica es también conocida como cortar el tiempo (*time slicing*), porque a cada proceso se le da una rodaja de tiempo antes de ser expulsado.

Con la planificación en turno rotatorio, el tema clave de diseño es la longitud del *quantum* de tiempo, o rodaja, a ser utilizada. Si el *quantum* es muy pequeño, el proceso se moverá por el sistema relativamente rápido. Por otra parte, existe una sobrecarga de procesamiento debido al manejo de la interrupción de reloj y por las funciones de planificación y activación. De esta forma, se deben evitar los *quantums* de tiempo muy pequeños. Una buena idea es que el *quantum* de tiempo debe ser ligeramente mayor que el tiempo requerido para una interacción o una función típica del proceso. Si es menor, muchos más procesos necesitarán, al menos, dos *quantums* de tiempo. La Figura 9.6 muestra el efecto que tiene en el tiempo de respuesta.

Nótese que en el caso extremo de un *quantum* de tiempo mayor que el proceso más largo en ejecución, la planificación en turno rotatorio degenera en FCFS.

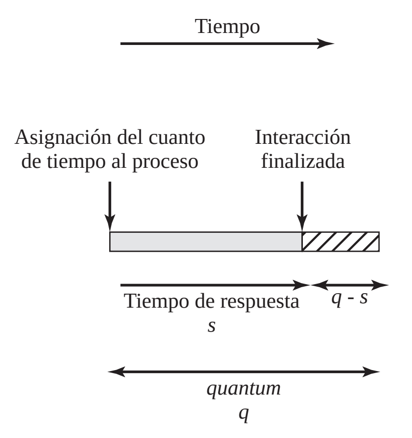
La Figura 9.5 y la Tabla 9.5 muestran los resultados de nuestro ejemplo utilizando *quantums* de tiempo q de 1 y 4 unidades de tiempo. Nótese que el proceso E, que es el trabajo más corto, obtiene mejoras significativas para un *quantum* de tiempo de 1.

La planificación en turno rotatorio es particularmente efectiva en sistemas de tiempo compartido de propósito general o en sistemas de procesamiento transaccional. Una desventaja de la planificación en turno rotatorio es que trata de forma desigual a los procesos limitados por el procesador y a los procesos limitados por la E/S. Generalmente, un proceso limitado por la E/S tiene ráfagas de procesador más cortas (cantidad de tiempo de ejecución utilizada entre operaciones de E/S) que los procesos limitados por el procesador. Si hay una mezcla de los dos tipos de procesos, sucederá lo siguiente: un proceso limitado por la E/S utiliza el procesador durante un periodo corto y luego se bloquea; espera a que complete la operación de E/S y a continuación se une a la cola de listos. Por otra parte, un proceso limitado por el procesador generalmente utiliza la rodaja de tiempo completa mientras ejecuta e inmediatamente vuelve a la cola de listos. De esta forma, los procesos limitados por el procesador tienden a recibir una rodaja no equitativa de tiempo de procesador, lo que conlleva un mal rendimiento de los procesos limitados por la E/S, uso ineficiente de los recursos de E/S y un incremento en la varianza del tiempo de respuesta.

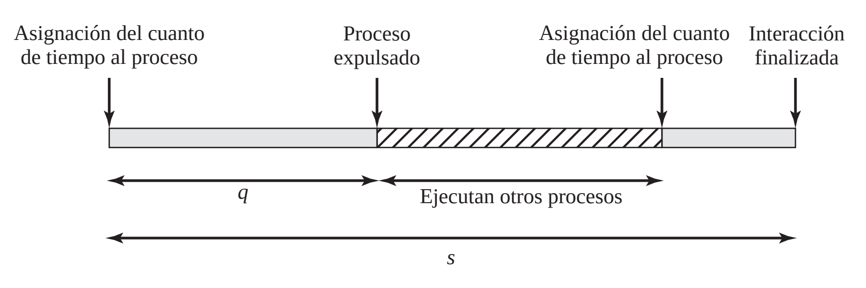
[HALD91] sugiere un refinamiento de la planificación en turno rotatorio que denomina turno rotatorio virtual (*virtual round robin* —VRR—) y que evita esta injusticia. La Figura 9.7 muestra un esquema. Los nuevos procesos que llegan se unen a la cola de listos, que es gestionada con FCFS. Cuando expira el tiempo de ejecución de un proceso, vuelve a la cola de listos. Cuando se bloquea un proceso por E/S, se une a la cola de E/S. Hasta aquí, todo como siempre. La nueva característica es

Tabla 9.5. Comparación de las políticas de planificación.

	Proceso	A	B	C	D	E	Media
	Tiempo de llegada	0	2	4	6	8	
	Tiempo de servicio (T_s)	3	6	4	5	2	
FCFS	Tiempo de finalización	3	9	13	18	20	
	Tiempo de estancia (T_r)	3	7	9	12	12	8.60
	T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$	Tiempo de finalización	4	18	17	20	15	
	Tiempo de estancia (T_r)	4	16	13	14	7	10.80
	T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$	Tiempo de finalización	3	17	11	20	19	
	Tiempo de estancia (T_r)	3	15	7	14	11	10.00
	T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN	Tiempo de finalización	3	9	15	20	11	
	Tiempo de estancia (T_r)	3	7	11	14	3	7.60
	T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT	Tiempo de finalización	3	15	8	20	10	
	Tiempo de estancia (T_r)	3	13	4	14	2	7.20
	T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN	Tiempo de finalización	3	9	13	20	15	
	Tiempo de estancia (T_r)	3	7	9	14	7	8.00
	T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$	Tiempo de finalización	4	20	16	19	11	
	Tiempo de estancia (T_r)	4	18	12	13	3	10.00
	T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$	Tiempo de finalización	4	17	18	20	14	
	Tiempo de estancia (T_r)	4	15	14	14	6	10.60
	T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63



(a) Cuanto de tiempo mayor que la interacción típica



(b) Cuanto de tiempo menor que la interacción típica

Figura 9.6. Efecto del tamaño del *quantum* de tiempo de expulsión.

una cola auxiliar FCFS a la que se mueven los procesos después de estar bloqueados en una E/S. Cuando se va a tomar una decisión de activación, los procesos en la cola auxiliar tienen preferencia sobre los de la cola de listos. Cuando se activa un proceso desde la cola auxiliar, éste ejecuta por un tiempo no superior a la rodaja de tiempo, menos el tiempo total que ha estado ejecutando desde que no está en la cola principal de listos. Los estudios de rendimiento realizados por los autores, indican que este enfoque es realmente superior al turno rotatorio en términos de equidad.

Primero el proceso más corto (*shortest process next*) Otro enfoque para reducir el sesgo a favor de los procesos largos inherente al FCFS es la política primero el proceso más corto (SPN). Es una política no expulsiva en la que se selecciona el proceso con el tiempo de procesamiento más corto esperado. De esta forma un proceso corto se situará a la cabeza de la cola, por delante de los procesos más largos.

La Figura 9.5 y la Tabla 9.5 muestran el resultado de nuestro ejemplo. Observe que el proceso E recibe servicio mucho antes que con FCFS. El rendimiento global también se mejora significativamente en términos del tiempo de respuesta. Sin embargo, se incrementa la variabilidad de los tiempos de respuesta, especialmente para los procesos más largos, y de esta forma se reduce la predecibilidad.

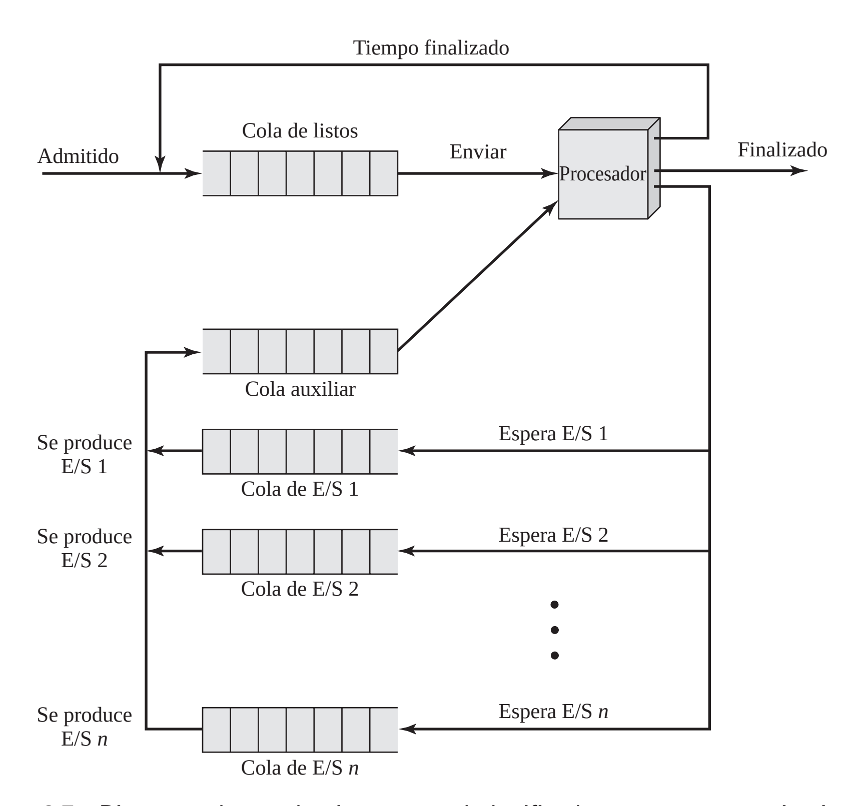


Figura 9.7. Diagrama de encolamiento para el planificador en turno rotatorio virtual.

Un problema de la política SPN es la necesidad de saber, o al menos de estimar, el tiempo de procesamiento requerido por cada proceso. Para trabajos por lotes, el sistema puede requerir que el programador estime el valor y se lo proporcione al sistema operativo. Si la estimación del programador es mucho menor que el tiempo actual de ejecución, el sistema podría abortar el trabajo. En un entorno de producción, ejecutan frecuentemente los mismos trabajos y, por tanto, se pueden recoger estadísticas. Para procesos interactivos, el sistema operativo podría guardar una media del tiempo de ejecución de cada «ráfaga» de cada proceso. La forma más sencilla de cálculo podría ser la siguiente:

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i \tag{9.1}$$

donde,

T_i = tiempo de ejecución del procesador para la instancia *i*-ésima de este proceso (tiempo total de ejecución para los trabajos por lotes; tiempo de ráfaga de procesador para los trabajos interactivos)

S_i = valor predicho para la instancia *i*-ésima

S_1 = valor predicho para la primera instancia; no calculado

Para evitar volver a calcular la suma completa cada vez, podemos reescribir la ecuación como

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n \tag{9.2}$$

Observe que esta fórmula otorga el mismo peso a cada instancia. Así, nos gustaría otorgar mayor peso a las instancias más recientes, ya que hay más probabilidades de que reflejen comportamientos futuros. El promedio exponencial es una técnica común de predecir valores futuros basándose en una serie de tiempos pasados:

$$S_{n+1} = \alpha\, T_n + (1 - \alpha)S_n \tag{9.3}$$

donde α es un factor de multiplicación constante ($0 < \alpha < 1$) que determina el peso relativo dado a las observaciones más recientes en relación a las antiguas observaciones. Comparar con la Ecuación (9.2). A través del uso de un valor constante de α, independientemente del número de observaciones pasadas, todos los valores pasados se consideran, pero los más distantes tienen menor peso. Para ver esto con mayor claridad, considere el siguiente desarrollo de la Ecuación (9.3):

$$S_{n+1} = \alpha T_n + (1 - \alpha)\, \alpha T_{n-1} + \ldots + (1 - \alpha)^i \alpha T_{n-1} + \ldots + (1 - \alpha)^n S_1 \tag{9.4}$$

Debido a que α y (1 - α) son menores que 1, cada término sucesivo de la anterior ecuación es más pequeño. Por ejemplo, para $\alpha = 0{,}8$, la Ecuación (9.4) queda

$$S_{n+1} = 0{,}8T_n + 0{,}16T_{n-1} + 0{,}032T_{n-2} + 0{,}0064T_{n-3} + \ldots$$

Más antigua sea la observación, menos cuenta en el promedio.

En la Figura 9.8 se muestra el tamaño del coeficiente como una función de su posición en el desarrollo. Mayor sea al valor de α, mayor será el peso dado a las observaciones más recientes. Para $\alpha = 0{,}8$, prácticamente se da todo el peso a las cuatro observaciones más recientes, mientras que para $\alpha = 0{,}2$, el promedio se extiende más o menos sobre las ocho últimas observaciones. La ventaja de utilizar un valor de α cercano a 1 es que el promedio reflejará un cambio rápido en las cantidades observadas. La desventaja es que si hay un cambio brusco en los valores observados y luego vuelven a su valor medio anterior, el uso de un valor alto de α puede dar lugar a oscilaciones en el promedio.

La Figura 9.9 compara el promedio simple con el exponencial (para dos valores diferentes de α). En la Figura 9.9a, el valor observado comienza en 1, crece gradualmente hasta el valor de 10, y luego se mantiene allí. En la Figura 9.9b, el valor observado comienza en 20, baja gradualmente hasta 10, y

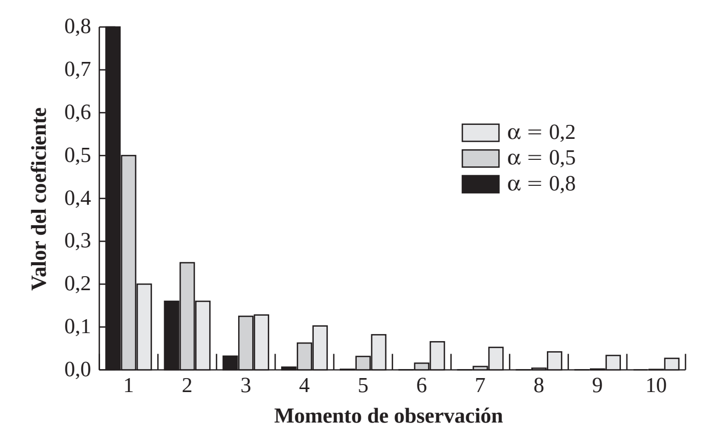
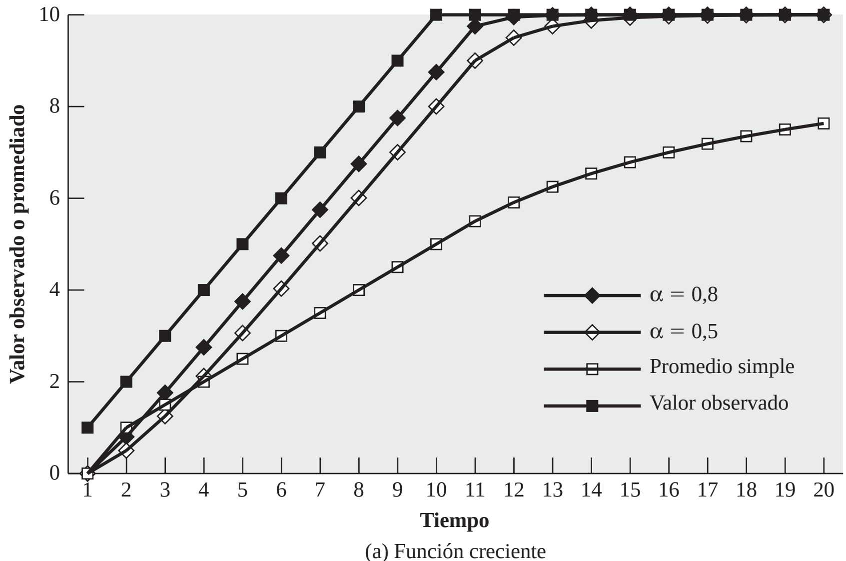
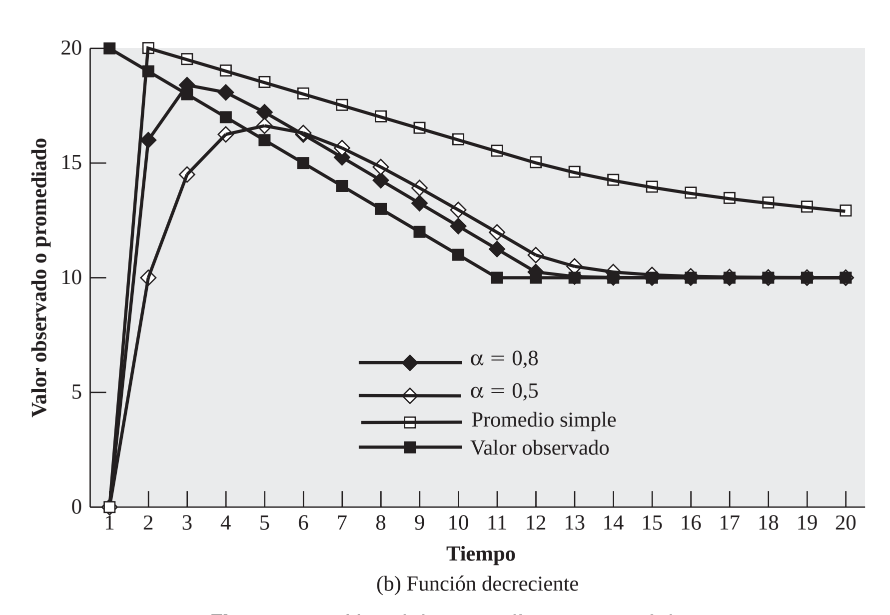


Figura 9.8. Coeficientes exponenciales suaves.



(a) Función creciente



(b) Función decreciente

Figura 9.9. Uso del promedio exponencial.

luego se mantiene allí. En ambos casos, comenzamos con un valor estimado de $S_1 = 0$. Esto da mayor prioridad a los nuevos procesos. Observar que el promedio exponencial reacciona a los cambios de comportamiento de los procesos más rápido que el promedio simple, y que los valores mayores de α generan reacciones más rápidas a los cambios en los valores observados.

Un riesgo con SPN es la posibilidad de inanición para los procesos más largos, si hay una llegada constante de procesos más cortos. Por otra parte, aunque SPN reduce la predisposición a favor de los trabajos más largos, no es deseable para un entorno de tiempo compartido o de procesamiento transaccional por la carencia de expulsión. Volviendo al peor caso descrito con FCFS, los procesos W, X, Y y Z seguirán ejecutando en el mismo orden, penalizando fuertemente al proceso corto Y.

Menor tiempo restante (*shortest remaining time*) La política del menor tiempo restante (SRT) es una versión expulsiva de SPN. En este caso, el planificador siempre escoge el proceso que tiene el menor tiempo de proceso restante esperado. Cuando un nuevo proceso se une a la cola de listos, podría tener un tiempo restante menor que el proceso actualmente en ejecución. Por tanto, el planificador podría expulsar al proceso actual cuando llega un nuevo proceso. Al igual que con SPN, el planificador debe tener una estimación del tiempo de proceso para realizar la función seleccionada, y existe riesgo de inanición para los procesos más largos.

SRT no tiene el sesgo en favor de los procesos largos que se puede encontrar en FCFS. A diferencia del turno rotatorio, no se generan interrupciones adicionales, reduciéndose la sobrecarga. Por otra parte, se deben almacenar los tiempos de servicio transcurridos, generando sobrecarga. SRT debería mejorar los tiempos de estancia de SPN, porque a un trabajo corto se le da preferencia sobre un trabajo más largo en ejecución.

Obsérvese que en nuestro ejemplo (Tabla 9.5), los tres procesos más cortos reciben servicio de forma inmediata, dando un tiempo de estancia normalizado para cada uno de ellos de 1,0.

Primero el de mayor tasa de respuesta (highest response ratio next) En la Tabla 9.5 hemos utilizado el tiempo de estancia normalizado, que es la tasa entre el tiempo de estancia y el tiempo actual de servicio, como un valor importante. Para cada proceso individual, nos gustaría minimizar esta tasa, y nos gustaría minimizar el valor medio sobre todos los procesos. En general, no podemos saber por adelantado el tiempo de servicio de los procesos, pero lo podemos aproximar, basándonos en la historia anterior, en alguna entrada de usuario o en un gestor de configuración. Considerar la siguiente tasa:

$$R = \frac{(w + s)}{s}$$

donde

R = tasa de respuesta

w = tiempo invertido esperando por el procesador

s = tiempo de servicio esperado

Si se despacha inmediatamente al proceso con este valor, R es igual al tiempo de estancia normalizado. Observar que el valor mínimo de R es 1,0, que sucede cuando un proceso acaba de entrar en el sistema.

De esta forma, nuestra regla de planificación es como sigue: cuando se completa o bloquea el proceso actual, elegir el proceso listo con el mayor valor de R. Este enfoque es atractivo por que tiene en cuenta la edad del proceso. Mientras que se favorece a los procesos más cortos (un menor denominador lleva a una mayor tasa), el envejecimiento sin servicio incrementa la tasa, por lo que un proceso más largo podría competir con los trabajos más cortos.

Como en SRT y SPN, el tiempo de servicio se debe estimar para usar la política primero el de mayor tasa de respuesta (HRRN).

Retroalimentación (*feedback*) Si no es posible averiguar el tiempo de servicio de varios procesos, SPN, SRT y HRRN no se pueden utilizar. Otra forma de establecer una preferencia para los trabajos más cortos es penalizar a los trabajos que han estado ejecutando más tiempo. En otras palabras, si no podemos basarnos en el tiempo de ejecución restante, nos podemos basar en el tiempo de ejecución utilizado hasta el momento.

La forma de hacerlo es la siguiente. La planificación se realiza con expulsión (por rodajas de tiempo), y se utiliza un mecanismo de prioridades dinámico. Cuando un proceso entra en el sistema, se sitúa en CL0 (*véase* Figura 9.4). Después de su primera expulsión, cuando vuelve al estado de Listo, se sitúa en CL1. Cada vez que es expulsado, se sitúa en la siguiente cola de menor prioridad. Un proceso corto se completará de forma rápida, sin llegar a migrar muy lejos en la jerarquía de colas de listos. Un proceso más largo irá degradándose gradualmente. De esta forma, se favorece a los procesos nuevos más cortos sobre los más viejos y largos. Dentro de cada cola, excepto en la cola de menor prioridad, se utiliza un mecanismo FCFS. Una vez en la cola de menor prioridad, un proceso no puede descender más, por lo que es devuelto a esta cola repetidas veces hasta que se consigue completar. De esta forma, esta cola se trata con una política de turno rotatorio.

La Figura 9.10 ilustra el mecanismo de planificación con retroalimentación mostrando el camino que seguirá un proceso a lo largo de las colas[5]. Este enfoque es conocido como **retroalimentación multinivel**, ya que el sistema operativo adjudica el procesador a un proceso y, cuando el proceso se bloquea o es expulsado, lo vuelve a situar en una de las varias colas de prioridad.

Hay una serie de variaciones de este esquema. Una versión sencilla consiste en realizar la expulsión de la misma manera que la política rotatoria: con intervalos periódicos. Nuestro ejemplo muestra esta situación (Figura 9.5 y Tabla 9.5) para una rodaja de una unidad de tiempo. Observar que en este caso, el comportamiento es similar a un turno rotatorio con rodaja de tiempo 1.

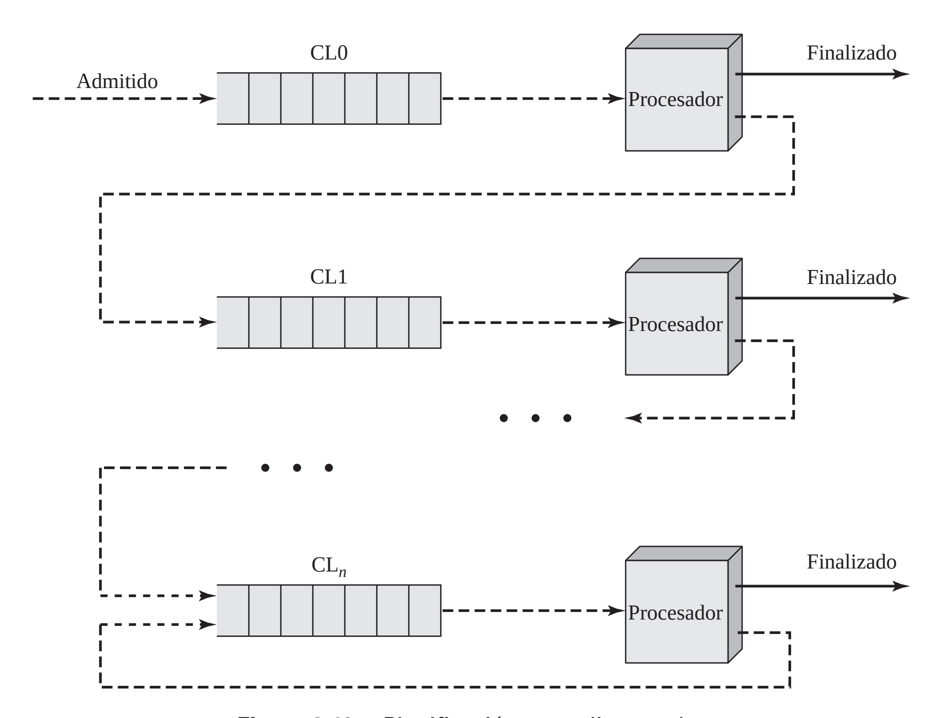


Figura 9.10. Planificación retroalimentada.

[5] Se utilizan las líneas de tiempo para enfatizar que se trata de un diagrama de secuencias de tiempo y no una representación estática de posibles transacciones, como en la Figura 9.4.

Un problema que presenta el esquema simple antes contado es que el tiempo de estancia para procesos más largos se puede alargar de forma alarmante. De hecho, puede ocurrir inanición si están entrando nuevos trabajos frecuentemente en el sistema. Para compensar este problema, podemos variar los tiempos de expulsión de cada cola: un proceso de la cola CL0 tiene una rodaja de una unidad de tiempo; un proceso de la cola CL1 tiene una rodaja de dos unidades de tiempo, y así sucesivamente. En general, a un proceso de la cola CL*i* se le permite ejecutar 2^i unidades de tiempo antes de ser expulsado. Este esquema se muestra en nuestro ejemplo de la Figura 9.5 y Tabla 9.5.

Incluso asignando rodajas de tiempo mayores a las colas de menor prioridad, un proceso grande puede sufrir inanición. Una posible solución es promover a los procesos a colas de mayor prioridad después de que pasen un determinado tiempo esperando servicio en su cola actual.

COMPARACIÓN DE RENDIMIENTO

Claramente, el rendimiento de las políticas de planificación es un factor crítico en su elección. Sin embargo, es imposible hacer una comparación definitiva, porque el rendimiento relativo dependerá de multitud de factores, que incluyen la distribución de probabilidad de los tiempos de servicio de varios procesos, la eficiencia de la planificación y del mecanismo del cambio de contexto, la naturaleza de las demandas de E/S y el rendimiento de los subsistemas de E/S. A pesar de todo, a continuación intentamos extraer algunas conclusiones generales.

Análisis de colas En esta sección haremos uso de las fórmulas básicas de colas, y supondremos que las llegadas siguen Procesos de Poisson y que los tiempos de servicio son exponenciales[6].

Primero, hacemos la observación de que cualquier disciplina de planificación que elija el siguiente elemento a ser servido de forma independiente al tiempo de servicio, cumple la siguiente relación:

$$\frac{T_r}{T_s} = \frac{1}{1-p}$$

donde,

T_r = Tiempo de estancia o residencia; tiempo total en el sistema, esperando y ejecutando

T_s = Tiempo medio de servicio; tiempo medio gastado en estado Ejecutando

ρ = Utilización de procesador

En particular, un planificador basado en prioridad, en que la prioridad de cada proceso se asigna de forma independiente del tiempo esperado de servicio, proporciona el mismo tiempo medio de estancia y tiempo medio de estancia normalizado que el enfoque FCFS. Además, la presencia o ausencia de expulsión no genera diferencia en estas medias.

Con la excepción de la planificación cíclica y FCFS, las diversas políticas de planificación consideradas en adelante, hacen sus selecciones basándose en los tiempos esperados de servicio. Por desgracia, resulta bastante complicado desarrollar modelos analíticos cercanos a estas disciplinas. Sin embargo, podemos hacernos una idea del rendimiento relativo de estos algoritmos, en comparación

---

[6] Se puede encontrar resumida la terminología de colas utilizada en este capítulo en el Apéndice 9B. En el Sitio “Computer Science Student Support” en WilliamStallings.com/StudentSupport.html se puede encontrar una introducción básica al análisis de colas.

Tabla 9.6. Fórmulas para colas de un único servidor con dos categorías de prioridades.

Suposiciones: 1. Tasa de llegada Poisson. 2. Los elementos con prioridad 1 se sirven antes que los elementos con prioridad 2. 3. Primero en llegar primero en servirse para los elementos con la misma prioridad. 4. No se interrumpe ningún elemento mientras está siendo atendido. 5. Ningún elemento abandona la cola (pierde llamadas retrasadas).	
a) Fórmulas Generales $\lambda = \lambda_1 + \lambda_2$ $\rho_1 = \lambda_1 T_{s1}; \rho_2 = \lambda_2 T_{s2}$ $\rho = \rho_1 + \rho_2$ $T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$ $T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$	
b) Sin interrupciones; tiempos de servicio exponencial	**c) Disciplina de colas expulsiva con reanudación; tiempos de servicio exponencial**
$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1}$ $T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}$	$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1}}{1 - \rho_1}$ $T_{r2} = T_{s2} + \frac{1}{1 - \rho_1}\left(\rho_1 T_{s2} + \frac{\rho T_s}{1 - \rho}\right)$

con FCFS, considerando una planificación con prioridades en la que la prioridad está basada en el tiempo de servicio.

Si la planificación se realiza basándose en la prioridad y si a los procesos se les asigna una prioridad basándose en el tiempo de servicio, entonces surgen las diferencias. La Tabla 9.6 muestra las fórmulas resultantes cuando se asumen dos clases de prioridad, con diferentes tiempos de servicio para cada clase. En la tabla, λ indica la tasa de llegada. Los resultados se pueden generalizar para cualquier número de clases de prioridad. Observar que las fórmulas son diferentes en la planificación con y sin expulsión. En el último caso se asume que se interrumpe a un proceso de menor prioridad cuando un proceso de mayor prioridad pasa a listo.

Como ejemplo, consideremos el caso de dos clases de prioridad, con el mismo número de llegadas de procesos en cada clase y con el tiempo medio de servicio para la clase de menor prioridad de 5 veces el tiempo de la clase de mayor prioridad. De esta forma, queremos dar preferencia a los procesos más cortos. La Figura 9.11 muestra el resultado total. Dando preferencia a los trabajos más cortos, se mejora el tiempo medio de estancia normalizado en los niveles de utilización superiores. Como sería posible esperar, la mejora es mayor con el uso de expulsión. Observar que, sin embargo, el rendimiento global no se ve muy afectado.

Sin embargo, surgen diferencias significativas cuando se consideran dos clases de prioridad por separado. La Figura 9.12 muestra el resultado para los procesos de mayor prioridad y menor tamaño. Para poder comparar, la línea superior del gráfico asume que no se usan prioridades, pero que estamos observando el rendimiento relativo de la mitad de los procesos que tienen menor tiempo de procesamiento. Las otras dos líneas asumen que se asigna a estos procesos una prioridad mayor. Las me-

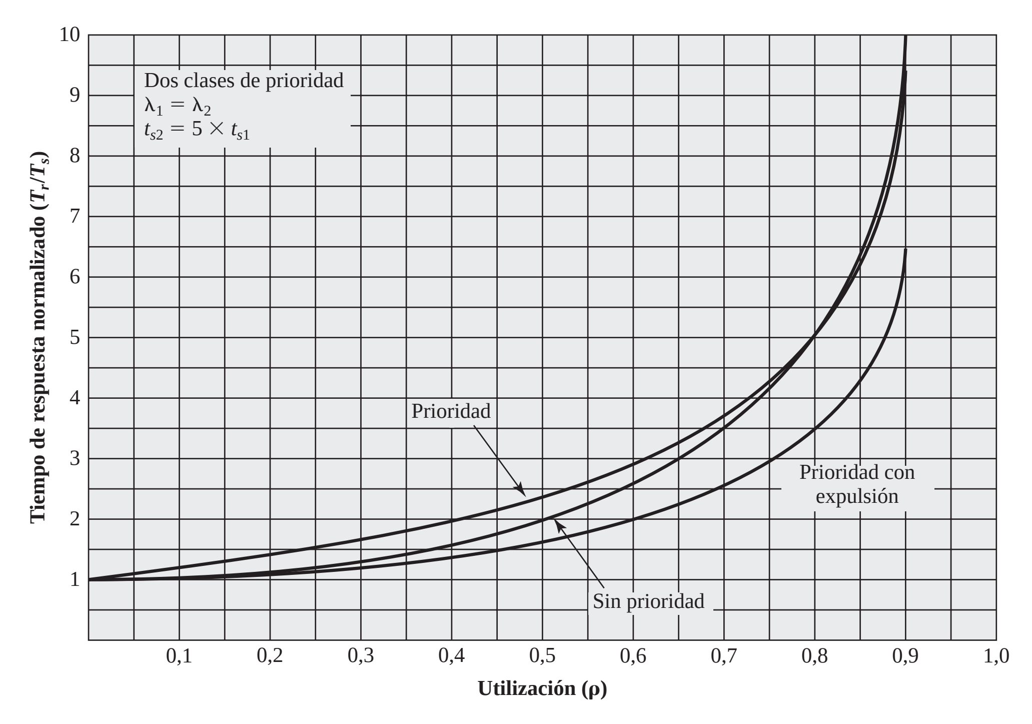


Figura 9.11. Tiempo total de respuesta normalizado.

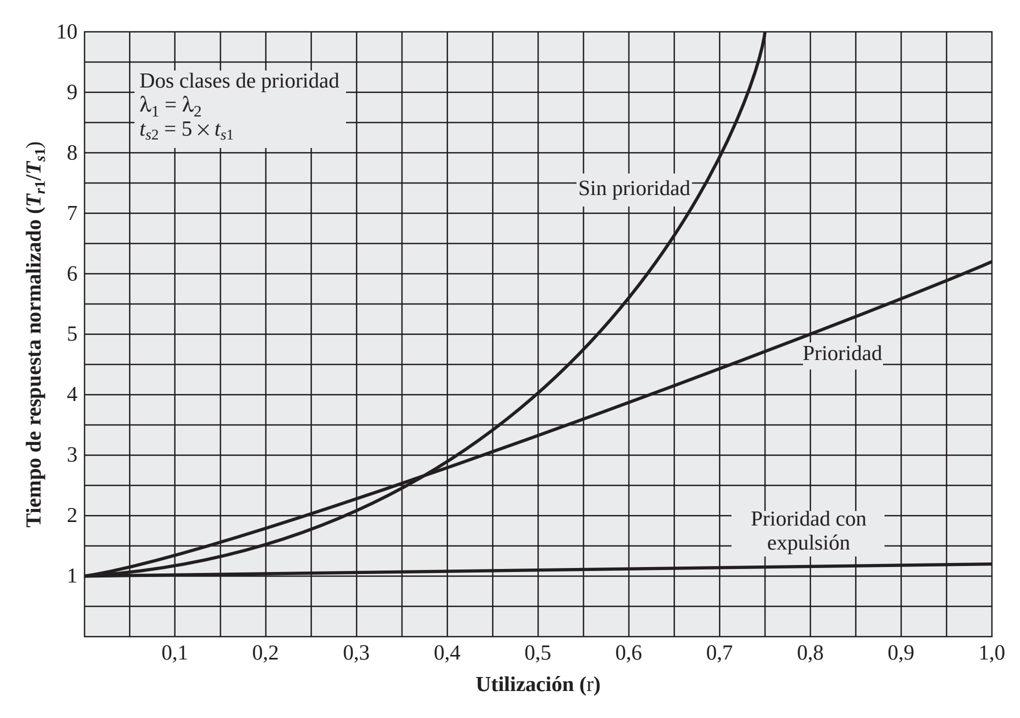


Figura 9.12. Tiempo de respuesta normalizado para los procesos más cortos.

joras son significativas cuando se ejecuta el sistema utilizando una planificación con prioridades sin expulsión. Estas mejoras son todavía más significativas cuando se utiliza expulsión.

La Figura 9.13 muestra el mismo análisis para los procesos de menor prioridad y mayor tamaño. Como cabe esperar, estos procesos sufren una degradación del rendimiento en planificaciones con prioridad.

Modelos de simulación Algunas dificultades del modelo analítico se pueden superar con el uso de la simulación discreta de casos, que permite modelar un amplio rango de políticas. La desventaja de la simulación es que los resultados de una determinada «ejecución» sólo se pueden aplicar a esa particular colección de procesos y bajo un conjunto particular de suposiciones. Sin embargo, se pueden obtener resultados útiles.

En [FINK88] se pueden encontrar los resultados de uno de estos estudios. La simulación consta de 50.000 procesos, con una tasa de llegada $\lambda = 0,8$ y un tiempo medio de servicio de $T_s = 1$. De esta forma, se supone que la utilización del procesador es $\rho = \lambda Ts = 0,8$. Por tanto, estamos midiendo solamente un punto de utilización. Para presentar los resultados se agrupan los procesos por percentiles de tiempo de servicio, cada uno de los cuales tiene 500 procesos. De esta forma, los 500 procesos con el menor tiempo de servicio están en el primer percentil; con éstos eliminados, los restantes 500 procesos con menor tiempo de servicio se incluyen en el segundo percentil; y así sucesivamente. Esto nos permite ver el efecto de diversas políticas como función de la longitud del proceso.

La Figura 9.14 muestra el tiempo de estancia normalizado, y la Figura 9.15 muestra el tiempo medio de espera. Mirando el tiempo de estancia, podemos observar que el rendimiento de FCFS es muy desfavorable, con la tercera parte de los procesos con un tiempo de estancia mayor a 10 veces el tiempo de servicio; además, son los procesos más cortos. Por otra parte, el tiempo de espera absoluto es uniforme como era de esperar, ya que la planificación es independiente del tiempo de ser-

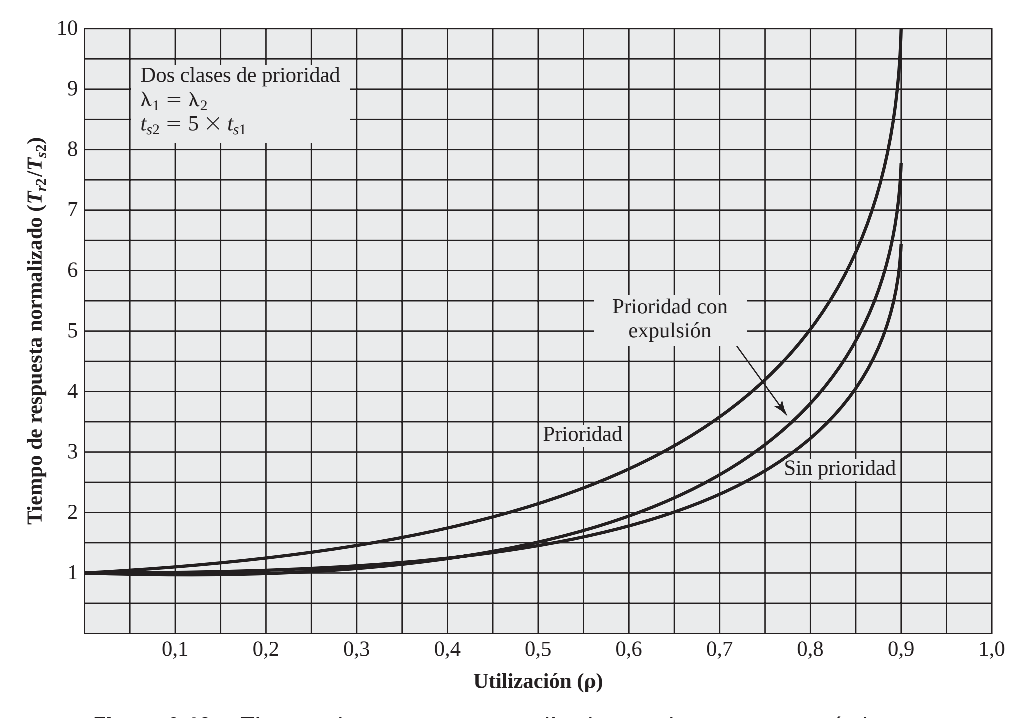


Figura 9.13. Tiempo de respuesta normalizado para los procesos más largos.

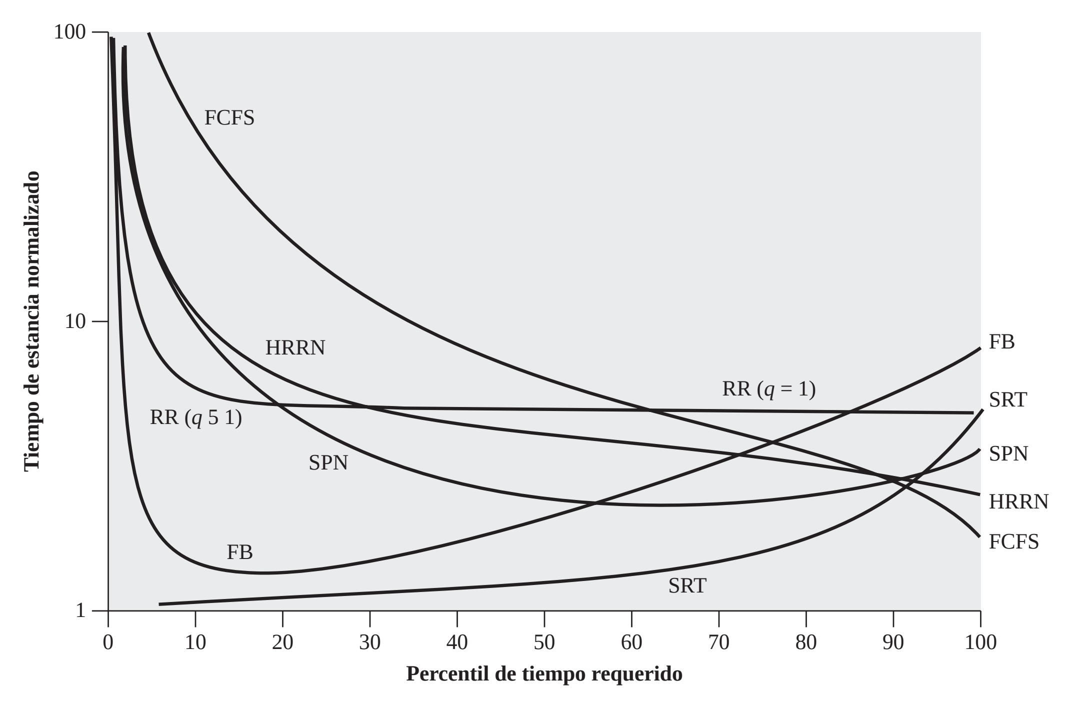


Figura 9.14. Resultados de la simulación para el tiempo de estancia normalizado.

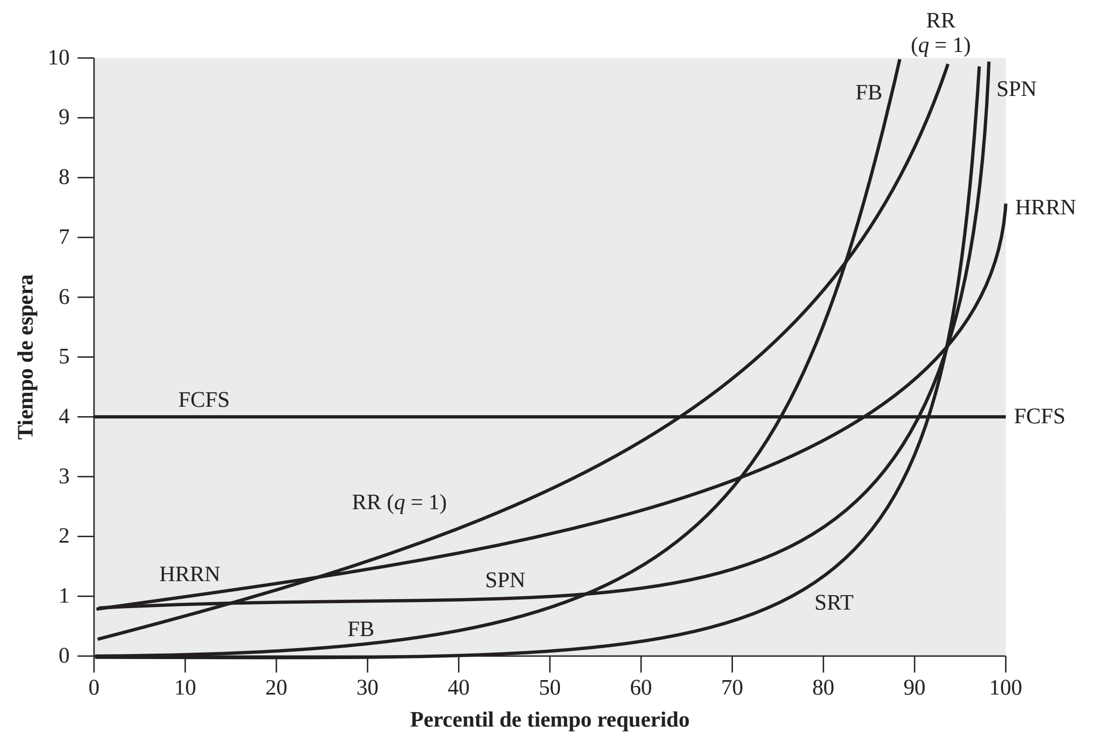


Figura 9.15. Resultados de la simulación para el tiempo de espera.

vicio. Las figuras muestran una planificación rotatoria con una rodaja de una unidad de tiempo. Excepto para los procesos más cortos, que ejecutan en menos de una rodaja, la planificación rotatoria da un tiempo de estancia normalizado de 5 para todos los procesos, tratando a todos de forma justa. Primero el proceso más corto, funciona mejor que la planificación de turno rotatorio, excepto para los procesos más cortos. El menor tiempo restante, la versión expulsiva de SPN, funciona mejor que SPN excepto para el 7% de los procesos más largos. Hemos visto que, entre las políticas no expulsivas, FCFS favorece a los procesos más largos y SPN favorece a los más cortos. Primero el de mayor tasa de respuesta, intenta ser un compromiso entre estos dos efectos, y realmente se confirma en las figuras. Por último, la figura muestra la planificación de retroalimentación con rodajas fijas y uniformes en cada cola de prioridad. Como es de esperar, FB funciona bastante bien para procesos cortos.

LA PLANIFICACIÓN CONTRIBUCIÓN JUSTA (FAIR-SHARE SCHEDULING)

Todos los algoritmos de planificación discutidos hasta el momento tratan a la colección de procesos listos como un simple conjunto de procesos de los cuales seleccionar el siguiente a ejecutar. Este conjunto de procesos se podría romper con el uso de prioridades, pero es homogéneo.

Sin embargo, en un sistema multiusuario, si las aplicaciones o trabajos de usuario se pueden organizar como múltiples procesos (o hilos), hay una estructura en la colección de procesos que no se reconoce en los planificadores tradicionales. Desde el punto de vista del usuario, la preocupación no es cómo ejecuta un simple proceso, sino cómo ejecutan su conjunto de procesos, que forman una aplicación. De esta forma, sería deseable tomar decisiones de planificación basándose en estos conjuntos de procesos. Este enfoque se conoce normalmente como planificación contribución justa. Además, el concepto se puede extender a un grupo de usuarios, incluso si cada usuario se representa con un solo proceso. Por ejemplo, en un sistema de tiempo compartido, podríamos querer considerar a todos los usuarios de un determinado departamento como miembros del mismo grupo. Se podrían tomar las decisiones de planificación intentando dar a cada grupo un servicio similar. De esta forma, si muchos usuarios de un departamento se meten en el sistema, nos gustaría ver cómo el tiempo de respuesta se degrada sólo para los miembros de este departamento, más que para los usuarios de otros departamentos.

El término *contribución justa* indica la filosofía que hay detrás de este planificador. A cada usuario se le asigna una prima de algún tipo que define la participación del usuario en los recursos del sistema como una fracción del uso total de dichos recursos. En particular, a cada usuario se le asigna una rodaja del procesador. Este esquema debería operar, más o menos, de forma lineal. Así, si un usuario A tiene el doble de prima que un usuario B, en una ejecución larga, el usuario A debería ser capaz de hacer el doble de trabajo que el usuario B. El objetivo del planificador contribución justa es controlar el uso para dar menores recursos a usuarios que se han excedido de su contribución justa y mayores recursos a los que no han llegado.

Se han realizado una serie de propuestas para los planificadores contribución justa [HENR84, KAY88, WOOD86]. En esta sección se describe el esquema propuesto por [HENR84], que ha sido implementado en diversos sistemas UNIX. A este sistema se le denomina FSS (*Fair Share Scheduler*). FSS considera el histórico de ejecución de un grupo de procesos relacionados, junto con el histórico de ejecución de cada uno de los procesos, para tomar decisiones de planificación. El sistema divide a la comunidad de usuarios en un conjunto de grupos y destina una fracción del procesador a cada grupo. De esta forma, podría haber 4 grupos, cada uno con un 25% del procesador. De hecho, a cada grupo se le proporciona un sistema virtual que ejecuta más lento que un sistema completo.

La planificación se realiza en base a la prioridad y tiene en cuenta la prioridad del proceso, su uso reciente de procesador y el uso reciente de procesador del grupo al que pertenece el proceso. Mayor

sea el valor numérico de la prioridad, menor será la prioridad. La siguiente fórmula se aplica al proceso j del grupo k:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = \text{Base}_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

donde

$CPU_j(i)$ = Medida de utilización del procesador por el proceso j en el intervalo i

$GCPU_k(i)$ = Medida de utilización del procesador del grupo k en el intervalo i

$P_j(i)$ = Prioridad del proceso j al comienzo del intervalo i; valores más pequeños equivalen a prioridades más altas

$Base_j$ = Prioridad base del proceso j

W_k = Prima asignada al grupo k, con la restricción que $0 < W_k \leq 1$ y $\sum_k W_k = 1$

A cada proceso se le asigna una prioridad base. La prioridad de un proceso disminuye a medida que el proceso utiliza el procesador y a medida que el grupo al que pertenece el proceso utiliza el procesador. En el caso de la utilización del grupo, se normaliza la media dividiendo por el peso de ese grupo. Mayor sea el peso asignado al grupo, su utilización afectará menos a su prioridad.

La Figura 9.16 es un ejemplo en el que el proceso A está en un grupo y los procesos B y C están en un segundo grupo. Cada grupo tiene una prima de 0,5. Se asume que todos los procesos están limitados por el procesador y están normalmente listos para ejecutar. Todos los procesos tienen una prioridad base de 60. La utilización del procesador se mide de la siguiente manera: el procesador se interrumpe 60 veces por segundo; durante cada interrupción se incrementa el campo del uso del procesador del proceso actualmente en ejecución, así como el campo del grupo correspondiente. Las prioridades se recalculan una vez por segundo.

En la figura, se planifica primero al proceso A. Al final del primer segundo se expulsa. Los procesos B y C tienen ahora la mayor prioridad, y se planifica al proceso B. Al final de la segunda unidad de tiempo, el proceso A tiene la mayor prioridad. Fijarse cómo se repite el patrón: el núcleo planifica los procesos en orden: A, B, A, C, A, B y así sucesivamente. De esta forma, el 50% del procesador se destina al proceso A, que constituye un grupo, y el otro 50% a los procesos B y C que constituyen otro grupo.

9.3. PLANIFICACIÓN UNIX TRADICIONAL

En esta sección examinamos la planificación tradicional UNIX, que se utiliza tanto en UNIX SVR3 como en 4.3 BSD UNIX. Estos sistemas están orientados a entornos interactivos de tiempo compartido. El algoritmo de planificación está diseñado para proporcionar buenos tiempos de respuesta a usuarios interactivos a la vez que asegura que los trabajos de fondo (*background*) de baja prioridad no sufran inanición. Aunque este sistema ha sido reemplazado en los sistemas UNIX modernos, merece la pena examinarlo porque es representativo de los algoritmos prácticos de planificación de tiem-

Tiempo	Proceso A			Proceso B			Proceso C		
	Prioridad	Contador CPU proceso	Contador CPU grupo	Prioridad	Contador CPU proceso	Contador CPU grupo	Prioridad	Contador CPU proceso	Contador CPU grupo
0	60	0 1 2 • • 60	0 1 2 • • 60	60	0	0	60	0	0
1	90	30	30	60	0 1 2 • • 60	0 1 2 • • 60	60	0	0 1 2 • • 60
2	74	15 16 17 • • 75	15 16 17 • • 75	90	30	30	75	0	30
3	96	37	37	74	15	15 16 17 • • 75	67	0 1 2 • • 60	15 16 17 • • 75
4	78	18 19 20 • • 78	18 19 20 • • 78	81	7	37	93	30	37
5	98	39	39	70	3	18	76	15	18

Grupo 1 (Proceso A) — Grupo 2 (Procesos B y C)

El rectángulo coloreado representa el proceso en ejecución

Figura 9.16. Ejemplo de planificación contribución justa – tres procesos, dos grupos.

po compartido. El esquema de planificación de SVR4 incluye una adaptación para requisitos de tiempo real, por lo que se verá en el Capítulo 10.

El planificador UNIX tradicional emplea una retroalimentación multinivel y utiliza planificación de turno rotatorio en cada una de las colas de prioridad. El sistema hace uso de expulsión de 1 segun-

do, es decir, si un proceso no se bloquea o se completa en un segundo, es expulsado. La prioridad está basada en el tipo de proceso y el histórico de ejecución. Se aplican las siguientes fórmulas:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = \text{Base}_j + \frac{CPU_j(i)}{2} + ajuste_j$$

donde

$CPU_j(i)$ = Medida de utilización del procesador por el proceso *j* en el intervalo *i*

$P_j(i)$ = Prioridad del proceso *j* al comienzo del intervalo *i*; valores más pequeños equivalen a prioridades más altas

$Base_j$ = Prioridad base del proceso *j*

$ajuste_j$ = Factor de ajuste controlable por el usuario

La prioridad de cada proceso se recalcula una vez por segundo, momento en el que se realiza una nueva decisión de planificación. El propósito de la prioridad base es dividir todos los procesos en bandas fijas de niveles de prioridad. Los componentes *CPU* y *ajuste* se limitan para impedir que los procesos migren de su banda asignada (asignada por el nivel de prioridad base). Estas bandas se utilizan para optimizar el acceso a dispositivos de bloques (por ejemplo, disco) y para permitir al sistema operativo responder rápidamente a llamadas al sistema. En orden decreciente de prioridad las bandas son

- Intercambiador (*swapper*)
- Control de dispositivos de bloque de E/S
- Manipulación de ficheros
- Control de dispositivos orientados a carácter de E/S
- Procesos de usuario

Esta jerarquía debería proporcionar el uso más eficiente de los dispositivos de E/S. Dentro de la banda de procesos de usuario, el uso del histórico de ejecución tiende a penalizar los procesos limitados por el procesador a costa de los procesos limitados por la E/S. De nuevo, esto debería mejorar la eficiencia. Junto con el esquema expulsivo cíclico, la estrategia de planificación está bien diseñada para satisfacer los requerimientos del tiempo compartido de propósito general.

En la Figura 9.17 se muestra un ejemplo de planificación de procesos. Los procesos A, B y C se crean al mismo tiempo con una prioridad base de 60 (ignoraremos el valor *ajuste*). El reloj interrumpe al sistema 60 veces por segundo e incrementa un contador para el proceso en ejecución. El ejemplo supone que ninguno de los procesos se bloquea y que ningún otro proceso está listo para ejecutar. Compárese con la Figura 9.16.

9.4. RESUMEN

El sistema operativo debe realizar tres tipos de decisiones de planificación respecto a la ejecución de procesos. La planificación a largo plazo determina cuándo se admiten nuevos procesos al sistema. La planificación a medio plazo es parte de la función de intercambio y determina cuándo un programa se

Tiempo	Proceso A		Proceso B		Proceso C	
	Prioridad	Contador CPU	Prioridad	Contador CPU	Prioridad	Contador CPU
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

El rectángulo coloreado representa el proceso en ejecución

Figura 9.17. Ejemplo de planificación tradicional de procesos UNIX.

trae parcial o totalmente a memoria principal para que pueda ser ejecutado. La planificación a corto plazo determina qué proceso listo será ejecutado por el procesador. Este capítulo se centra en los aspectos relativos a la planificación a corto plazo.

Para el diseño de un planificador a corto plazo se deben tener en cuenta una serie de criterios. Algunos de estos criterios están relacionados con el comportamiento del sistema tal y como lo percibe el usuario (orientados al usuario), mientras que otros miran la efectividad total del sistema para satisfacer las necesidades de todos los usuarios (orientados al sistema). Algunos criterios se relacionan específicamente con medidas cuantitativas de rendimiento, mientras que otros son de naturaleza más cualitativa. Desde el punto de vista del usuario el tiempo de respuesta es, generalmente, la característica más importante de un sistema, mientras que desde el punto de vista del sistema son importantes el rendimiento o la utilización del procesador.

Se han desarrollado varios algoritmos para tomar decisiones de planificación a corto plazo entre todos los procesos listos para ejecutar:

- **Primero en llegar, primero en servirse.** Selecciona el proceso que más tiempo ha estado esperando servicio.
- **Turno rotatorio.** Utiliza rodajas de tiempo para limitar los procesos en ejecución a una pequeña ráfaga de tiempo de ejecución, y rota entre todos los procesos listos.
- **Primero el proceso más corto.** Selecciona el proceso con el menor tiempo de procesamiento esperado y no expulsa a los procesos.
- **Menor tiempo restante.** Selecciona el proceso con el menor tiempo de procesamiento restante esperado. Un proceso puede ser expulsado cuando otro proceso pasa a listo.
- **Primero el de mayor tasa de respuesta.** Basa la decisión de planificación en una estimación del tiempo de estancia normalizado.
- **Retoalimentación:** Establece un conjunto de colas de planificación y sitúa los procesos en las colas basándose en su historia de ejecución y otros criterios.

La elección de algoritmo de planificación dependerá del rendimiento esperado y de la complejidad de la implementación.

9.5. LECTURAS RECOMENDADAS

Prácticamente todos los libros de texto de sistemas operativos cubren la planificación. En [KLEI04] y [CONW67] se presentan rigurosos análisis de colas de varias políticas de planificación. [DOWD93] proporciona un análisis de rendimiento muy instructivo de varios algoritmos de planificación.

CONW67 Conway, R.; Maxwell, W.; y Miller, L. *Theory of Scheduling*. Reading, MA: Addison-Wesley, 1967. Reprinted by Dover Publications, 2003.

DOWD93 Dowdy, L., y Lowery, C. P.S. *to Operating Systems*. Upper Saddle River,NJ: Prentice Hall, 1993.

KLEI04 Kleinrock, L. *Queuing Systems, Volume Three: Computer Applications*. New York: Wiley, 2004.

9.6. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

Activador	Primero en llegar, primero en servirse (FCFS)	Tiempo de estancia
Equidad	Previsibilidad	Tiempo de respuesta
Planificación a corto plazo	Promedio exponencial	Tiempo de residencia
Planificación a largo plazo	Rendimiento	Tiempo de servicio
Planificación a medio plazo	Retroalimentación multinivel	Turno rotatorio
Planificación con prioridades	Rodajas de tiempo	Utilización
Planificación contribución justa	Tasa de llegada	
Primero en entrar, primero en salir (FIFO)	Tiempo de espera	

CUESTIONES DE REPASO

9.1. Describa brevemente los tres tipos de planificación de procesos.

9.2. ¿Cuál es normalmente el requisito de rendimiento crítico en un sistema operativo interactivo?

9.3. ¿Cuál es la diferencia entre el tiempo de estancia y el tiempo de respuesta?

9.4. Para planificación de procesos, un valor bajo de prioridad, ¿representa una prioridad baja o una prioridad alta?

9.5. ¿Cuál es la diferencia entre planificación expulsiva y no expulsiva?

9.6. Defina brevemente la planificación FCFS.

9.7. Defina brevemente la planificación en turno rotatorio.

9.8. Defina brevemente la planificación primero el proceso más corto.

9.9. Defina brevemente la planificación menor tiempo restante.

9.10. Defina brevemente la planificación primero el de mayor tasa de respuesta.

9.11. Defina brevemente la planificación retroalimentación.

PROBLEMAS

9.1. Considere el siguiente conjunto de procesos

Nombre del proceso	Tiempo de llegada	Tiempo de proceso
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Realice el mismo análisis que el realizado en la Tabla 9.5 y en la Figura 9.5 para este conjunto.

9.2. Repita el Problema 9.1 con el siguiente conjunto de datos

Nombre del proceso	Tiempo de llegada	Tiempo de proceso
A	0	1
B	1	9
C	2	1
D	3	9

9.3. Demuestre que, entre los algoritmos de planificación no expulsivos, SPN proporciona el mínimo tiempo de espera medio para un conjunto de trabajos que llega al mismo tiempo. Asuma que el planificador debe ejecutar siempre una tarea si hay alguna disponible.

9.4. Suponga el siguiente patrón de ráfagas para un proceso: 6, 4, 6, 4, 13, 13, 13, y asuma que la conjetura inicial es 10. Realice un dibujo similar al de la Figura 9.9.

9.5. Considere el siguiente par de fórmulas como alternativa a la Ecuación (9.3):

$$S_{n+1} = \alpha T_n + (1 - \alpha)S_n$$

$$X_{n+1} = \min[Ubound, \max[Lbound, (\beta S_{n+1})]]$$

donde *Ubound* y *Lbound* son los límites superior e inferior preelegidos en el valor estimado de *T*. Se utiliza el valor de X_{n+1} en lugar del valor S_{n+1} para el algoritmo primero el proceso más corto. ¿Qué función realizan α y β y cuál es el efecto de mayores y menores valores en cada uno de ellos?

9.6. En el último ejemplo de la Figura 9.5, el proceso A ejecuta 2 unidades de tiempo antes de que se pase el control al proceso B. Otro posible escenario sería que A ejecutara 3 unidades de tiempo antes de que se pase el control al proceso B. ¿Qué variaciones en la política del algoritmo de planificación con retroalimentación tendrían en cuenta los dos posibles escenarios?

9.7. En un sistema uniprocesador no expulsivo, la cola de listos contiene 3 trabajos en el instante *t*, inmediatamente después de la finalización de un trabajo. Estos trabajos llegaron en los tiempos t_1, t_2 y t_3 y tienen tiempos estimados de ejecución r_1, r_2 y r_3, respectivamente. La Figura 9.18 muestra el incremento lineal de sus tasas de respuesta sobre el tiempo. Utilice este ejemplo para encontrar una variante a la planificación tasa de respuesta, conocida como planificación tasa de respuesta *minimax*, que minimice la máxima tasa de respuesta para un conjunto de trabajos ignorando futuras llegadas. (*Sugerencia*: decida primero qué trabajo planificar el último).

9.8. Demuestre que el algoritmo tasa de respuesta *minimax* del anterior problema minimiza la máxima tasa de respuesta de un lote de trabajos dado. (*Sugerencia*: centre su atención en el trabajo que tendrá la mayor tasa de respuesta y todos los trabajos ejecutados antes que él. Considere el mismo subconjunto de trabajos planificados en cualquier otro orden y observe la tasa de respuesta del trabajo que se ejecuta en último lugar. Observe que este subconjunto no debería ser mezclado con otros trabajos del conjunto total).

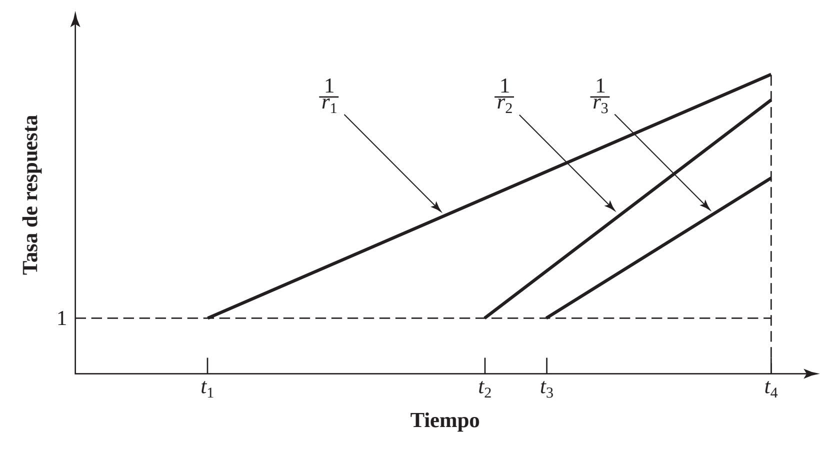


Figura 9.18. Tasa de respuesta como función del tiempo.

9.9. Defina el tiempo de residencia T_r como el tiempo total medio que un proceso pasa esperando y siendo atendido. Muestre cómo para FIFO, con tiempo de servicio medio T_s, tenemos $T_r = T_s /(1- \rho)$, donde ρ es la utilización.

9.10. Se multiplexa, sin ninguna sobrecarga, un procesador a velocidad infinita entre todos los procesos presentes en una cola de listos. (Esto es un modelo idealizado de planificación cíclica entre procesos listos utilizando rodajas de tiempo que son muy pequeñas en comparación con el tiempo de servicio medio). Muestre que para entradas Poisson desde una fuente infinita con tiempos de servicio exponenciales, el tiempo de respuesta medio R_x de un proceso con tiempo de servicio x viene dado por $R_x = x/(1- \rho)$. (*Sugerencia*: revise las ecuaciones básicas de colas en el documento Análisis de Colas en **WilliamsStallings.com/StudentSupport.html**. A continuación, considere el número de elementos esperando, *w*, en el sistema cuando llega un proceso dado).

9.11. La mayor parte de las planificaciones de turno rotatorio utilizan una rodaja de tiempo fija. De un argumento a favor de las rodajas pequeñas. Ahora de un argumento a favor de las rodajas grandes. Compare y contraste los tipos de sistema y trabajos a los que se aplican los argumentos. ¿Hay algunos sistemas o trabajos para los que ambos tamaños sean razonables?

9.12. En un sistema de colas, los nuevos trabajos deben esperar antes de ser atendidos. Mientras que un trabajo espera, su prioridad se incrementa de forma lineal con el tiempo empezando en cero y con una tasa α. El trabajo espera hasta que su prioridad alcanza la prioridad de los trabajos que están siendo atendidos; a continuación comienza a compartir el procesador con el resto de los trabajos de forma cíclica y su prioridad se continúa incrementando a una tasa β menor. Este algoritmo se denomina turno rotatorio egoísta, porque los trabajos intentan (en vano) monopolizar el procesador incrementando de forma constante su prioridad. Utilice la figura 9.19 para demostrar que el tiempo medio de respuesta R_x para un trabajo con tiempo de servicio x viene dado por

$$R_x = \frac{s}{1-\rho} + \frac{x-s}{1-\rho'}$$

donde

$$\rho = \lambda s \qquad \rho' = \rho\left(1 - \frac{\beta}{\alpha}\right) \qquad 0 \le \beta < \alpha$$

asumiendo que los tiempos de llegada y servicio son distribuciones exponenciales con media $1/\lambda$ y *s*, respectivamente. (*Sugerencia*: considere al sistema global y a los dos subsistemas de forma separada).

9.13. Un sistema interactivo con planificación rotatoria e intercambio intenta dar respuesta a peticiones sencillas de la siguiente manera: después de completar un turno rotatorio entre todos los procesos listos, el sistema determina la siguiente rodaja de tiempo que asignar a cada proceso listo en el siguiente turno dividiendo el tiempo máximo de respuesta por el número de proceso que requieren ser atendidos. ¿Es esta una política razonable?

9.14. ¿Qué tipo de proceso se favorece normalmente en una planificación de colas retroalimentadas multinivel, los procesos limitados por el procesador o los procesos limitados por la E/S? Explique brevemente por qué.

9.15. En un proceso de planificación basado en prioridades, el planificador sólo pasa el control a un proceso determinado si ningún otro proceso de mayor prioridad está en el estado de listo. Asuma que no se utiliza ninguna información adicional en el proceso de decisión. Asu-

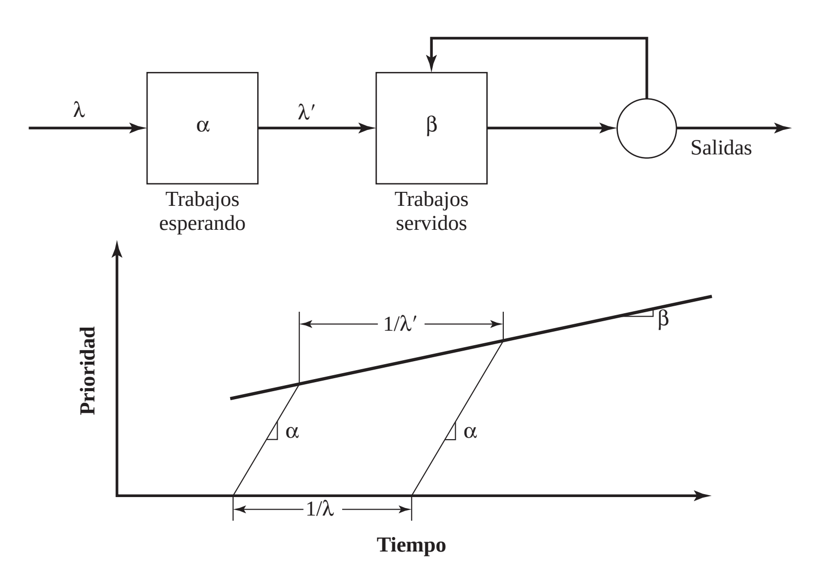


Figura 9.19. Planificación turno rotatorio egoísta.

ma también que las prioridades de los procesos se establecen en el momento de la creación del proceso y que no cambian. En un sistema que funcione con estos supuestos, ¿por qué es peligroso el uso de la Solución de Dekker (*véase* Sección A.1) para el problema de la exclusión mutua? Explíquelo justificando qué evento no deseable podría suceder y cómo podría suceder.

9.16. Cinco trabajos por lotes, del A al E, llegan a un centro de computación al mismo tiempo. Tienen tiempos estimados de ejecución de 15, 9, 3, 6 y 12 minutos respectivamente. Sus prioridades (definidas externamente) son 6, 3, 7, 9 y 4 respectivamente, con los valores menores correspondiendo a prioridades mayores. Para cada uno de los siguientes algoritmos de planificación, determine el tiempo de estancia de cada proceso y el tiempo de estancia medio de todos los trabajos. Ignore el tiempo de intercambio entre procesos. Explique cómo llega a las respuestas. En los últimos tres casos, asuma que sólo ejecuta un trabajo hasta que termina y que todos los trabajos están totalmente limitados por el procesador.

a) Planificación de turno rotatorio con una rodaja de 1 minuto

b) Planificación por prioridades

c) FCFS (ejecutan en orden 15, 9, 3, 6 y 12)

d) Primero el proceso más corto

APÉNDICE 9A TIEMPO DE RESPUESTA

El tiempo de respuesta es el tiempo que le lleva al sistema reaccionar frente a una determinada entrada. En una transacción interactiva, se podría definir como el tiempo entre la última pulsación de una tecla por parte del usuario y el comienzo de la visualización de los resultados por parte del computador. Para diferentes tipos de aplicaciones se podrían dar diferentes definiciones. En general, es el tiempo que le lleva al sistema responder a una petición de una determinada tarea.

De forma ideal, nos gustaría que el tiempo de respuesta a cualquier aplicación sea corto. Sin embargo, casi de forma fija, menores tiempos de respuesta implican mayores costes. Estos costes vienen de dos partes:

- **Potencia de computación.** Más rápido sea el procesador, menor será el tiempo de respuesta. Por supuesto, incrementar la potencia de computación significa incrementar los costes.
- **Requisitos de competencia.** Proporcionar un tiempo de respuesta rápida a algunos procesos puede penalizar a otros procesos.

De esta forma, se debe llegar a un compromiso entre el nivel del tiempo de respuesta y el coste que conlleva.

La Tabla 9.7, basada en [MART88], enumera seis rangos de tiempos de respuesta. Nos podemos enfrentar con dificultades de diseño cuando se requiere un tiempo de respuesta menor que 1 segundo. En sistemas que controlan o interactúan con una actividad externa, como una cadena de montaje, existe el requisito de tiempos de respuesta por debajo de un segundo. Cuando consideramos una interacción hombre-máquina, tal como una aplicación de entrada de datos, estamos en el dominio de tiempos de respuesta conversacionales. En este caso, sigue existiendo el requisito de tiempos cortos de respuesta, pero puede ser difícil evaluar la longitud de tiempo aceptable.

Tabla 9.7. Rangos de tiempos de respuesta.

Mayores de 15 segundos

Esto descarta interacciones conversacionales. Para determinado tipo de aplicaciones, a cierto tipo de usuarios no les importaría esperar más de 15 segundos una respuesta. Sin embargo, para una persona ocupada, esta cifra es intolerable. Si tenemos estos tiempos, se debe diseñar el sistema de forma que el usuario pueda cambiar a otras actividades y mire la respuesta más adelante.

Mayores de 4 segundos

Son normalmente demasiado largos para una conversación, ya que el operador debe mantener en memoria la información. Estos retardos dificultan mucho la solución de problemas y son frustrantes en actividades de entrada de datos. Sin embargo, tras la conclusión de cierta actividad principal, como al finalizar una transacción, estos retardos son tolerables.

De 2 a 4 segundos

Estos retrasos pueden dificultar las operaciones de terminal, que requieren una alta concentración. Esperar de 2 a 4 segundos en un terminal puede resultar sorprendentemente largo cuando el usuario está absorto en la tarea que está realizando. De nuevo, se puede aceptar un retardo en este rango tras la conclusión de una actividad menor.

Menos de 2 segundos

Cuando el usuario del terminal tiene que recordar información durante varias respuestas, el tiempo de respuesta tiene que ser corto. Más detalle tenga la información a recordar, mayor será la necesidad de tiempos de respuesta menores de 2 segundos. Para actividades elaboradas de terminal, 2 segundos representan un límite de tiempo importante.

Tiempos de respuesta por debajo de un segundo

Ciertos tipos de trabajos que requieren mucha concentración, especialmente en aplicaciones gráficas, requieren tiempos de respuesta muy cortos para mantener el interés y la atención del usuario durante largos periodos de tiempo.

Tiempos de respuesta de una décima de segundo

Actividades como presionar una tecla y ver el carácter en pantalla o seleccionar un elemento en la pantalla, deben ser casi instantáneas. menos de 0,1 segundos después de la acción. La interacción con un ratón requiere interacciones extremadamente rápidas.

La clave de la productividad en aplicaciones interactivas es un tiempo de respuesta rápido, como se ha confirmado en diversos estudios [SHNE84; THAD81; GUYN88]. Estos estudios muestran que cuando un ordenador y un usuario interactúan de forma que se asegure que ninguno debe esperar por el otro, la productividad se incrementa significativamente, el coste de los trabajos realizados en el computador se reduce y la calidad tiende a mejorar. Solía estar ampliamente aceptado que tiempos de respuesta relativamente lentos, hasta de 2 segundos, eran aceptables para la mayor parte de aplicaciones interactivas, porque la persona estaba pensando en la siguiente tarea a realizar. Sin embargo, ahora parece que la productividad se incrementa si se logran tiempos de respuesta rápidos.

Los resultados obtenidos sobre los tiempos de respuesta están basados en un análisis de transacciones *on-line*. Una transacción consiste en un mandato de usuario desde un terminal incluyendo la respuesta del sistema. Se puede dividir en dos secuencias de tiempo:

- Tiempo de respuesta del usuario: el tiempo que transcurre desde que el usuario recibe una respuesta completa a un mandato hasta que introduce el siguiente mandato. La gente suele referirse a este tiempo como tiempo de pensar.
- Tiempo de respuesta del sistema: el tiempo que transcurre desde que el usuario introduce un mandato hasta que la respuesta completa se presenta en el terminal.

Como ejemplo del efecto de reducir el tiempo de respuesta del sistema, la Figura 9.20 muestra los resultados de un estudio llevado a cabo con ingenieros que utilizan un programa de diseño asistido por computador para el diseño de circuitos integrados y tarjetas [SMIT83]. Cada transacción consiste en un mandato que altera de alguna forma el gráfico mostrado en pantalla. El resultado muestra que el número de transacciones se incrementa a medida que el tiempo de respuesta del sistema disminuye. Y aumenta drásticamente cuando el tiempo de respuesta del sistema cae por debajo de 1 segundo. Lo que sucede es que a medida que se reduce el tiempo de respuesta del sistema, también lo hace el tiempo de respuesta del usuario. Esto está relacionado con el efecto de la memoria a corto plazo y la atención humana.

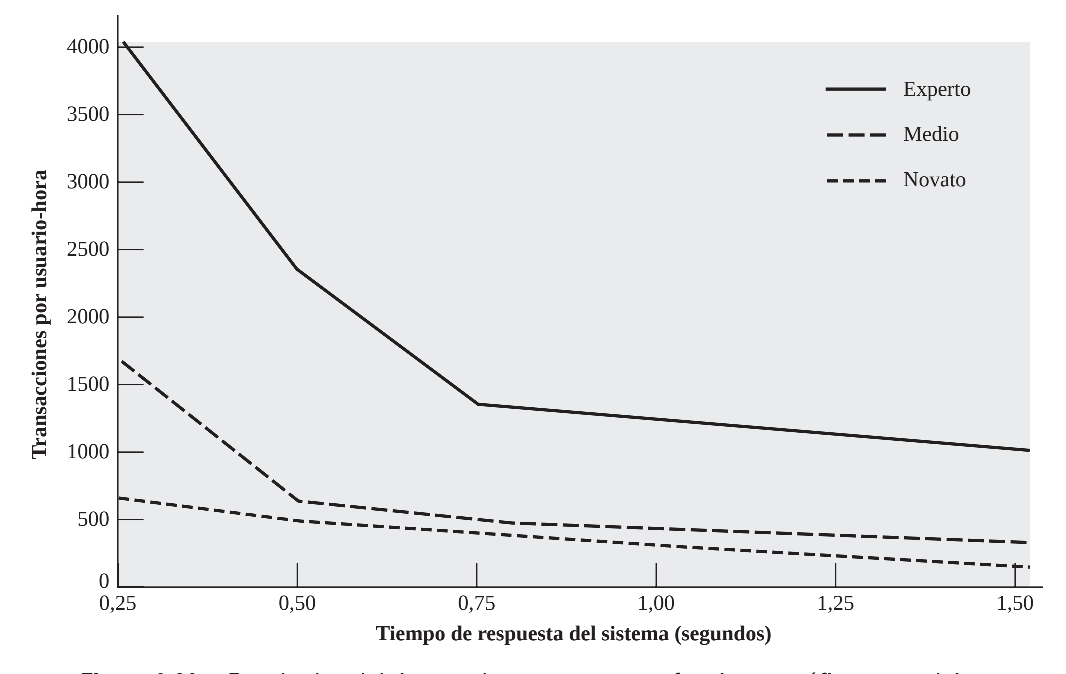


Figura 9.20. Resultados del tiempo de respuesta para funciones gráficas complejas.

Otra área donde el tiempo de respuesta se está volviendo crítico es la Web, ya sea en Internet o sobre una intranet. El tiempo que lleva que una página web aparezca en la pantalla del usuario puede variar mucho. Los tiempos de respuesta se pueden medir en función del nivel de la participación del usuario en la sesión; en particular, los sistemas con tiempos de respuesta muy rápidos tienden a centrar mejor la atención del usuario. Como se muestra en la Figura 9.21 [SEVC96], los sistemas web con un tiempo de respuesta menor o igual a 3 segundos, mantienen un alto nivel de atención del usuario. Como la Figura 9.21 indica [SEVC96], los sistemas web con tiempos de respuesta de 3 segundos o menos mantienen un alto nivel de atención del usuario. Con un tiempo de respuesta entre 3 y 10 segundos, se pierde algo de concentración, y un tiempo de respuesta mayor de 10 segundos desmoraliza al usuario, que cancela la sesión.

APÉNDICE 9B SISTEMAS DE COLAS

En este capítulo, y algunos capítulos posteriores, se utiliza algo de la teoría de colas. En este apéndice se presenta una breve definición de los sistemas de colas y se definen los conceptos clave. Para los lectores que no estén familiarizados con los sistemas de colas, hay una introducción básica en el Sitio «Computer Science Student Resource» en **WilliamStallings.com/StudentSupport.html.**

¿POR QUÉ ANÁLISIS DE COLAS?

A menudo es necesario hacer pronósticos del rendimiento en función de la carga existente o en función de la carga esperada en un nuevo entorno. Son posibles varios enfoques:

1. Hacer un análisis posterior, basándose en los valores actuales.
2. Hacer un pronóstico sencillo, extrapolando los valores conocidos al entorno esperado en el futuro.

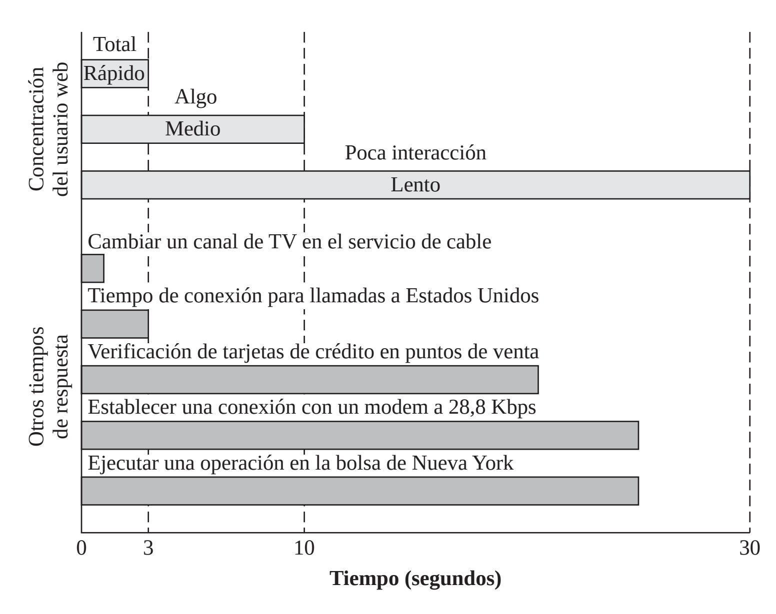


Figura 9.21. Requisitos de los tiempos de respuesta.

3. Desarrollar un modelo analítico, basado en el sistema de colas.
4. Programar y ejecutar un modelo de simulación.

La primera opción no es una opción en absoluto: esperar y ver qué sucede. Esto genera usuarios infelices y compras imprudentes. La segunda opción parece más prometedora. El analista debe tomar la posición de que es imposible predecir la demanda futura con cierto grado de certeza. Por tanto, no merece la pena intentar ningún modelado exacto. Por el contrario, un pronóstico preliminar proporcionará una estimación dentro del rango de lo posible. El problema de este enfoque es que el comportamiento de la mayor parte de los sistemas con una carga cambiante no es el que intuitivamente cabría esperar. Si hay un entorno en el que existe un recurso compartido (por ejemplo, una red, una línea de transmisión, un sistema de tiempo compartido) entonces, el rendimiento de ese sistema, normalmente responde de forma exponencial a los incrementos de la demanda.

La Figura 9.22 es un ejemplo representativo. La línea superior muestra lo que suele suceder con los tiempos de respuesta de los usuarios en un recurso compartido a medida que se incrementa la carga. La carga se expresa como una fracción de la capacidad. De esta forma, si estamos trabajando con un encaminador que es capaz de procesar y remitir 1000 paquetes por segundo, entonces una carga de 0,5 representa una tasa de llegada de 500 paquetes por segundo, y el tiempo de respuesta es la cantidad de tiempo que lleva retransmitir cualquier paquete entrante. La línea inferior es un pronóstico simple[7] basado en el conocimiento del comportamiento del sistema con una carga de 0,5. Es impor-

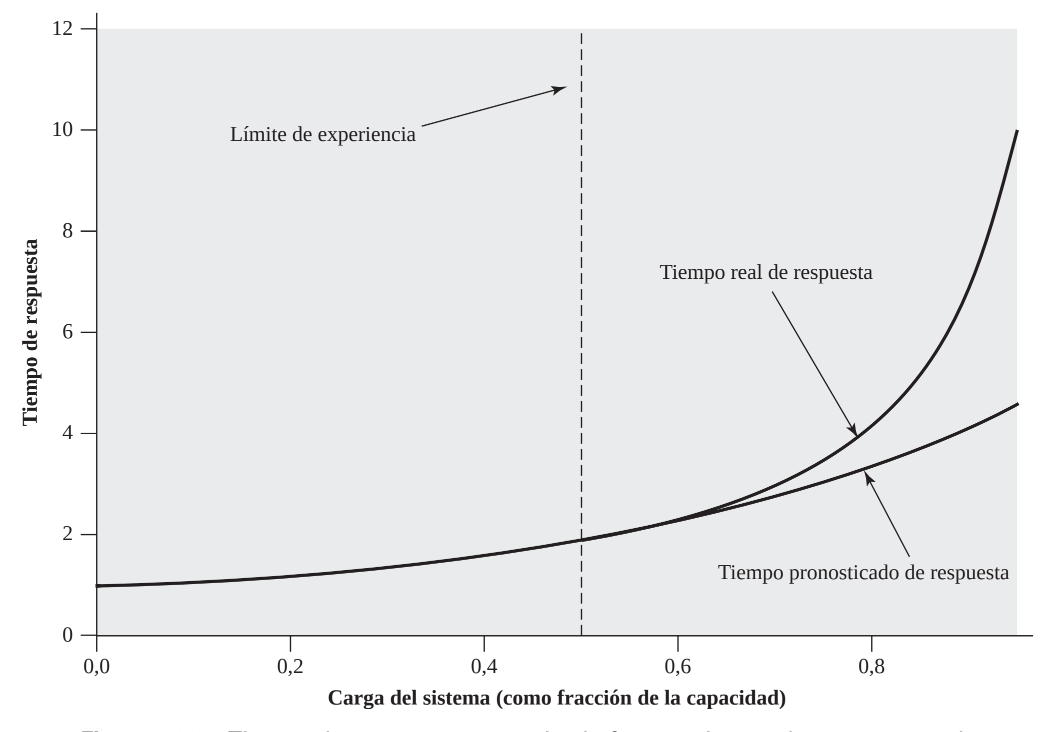


Figura 9.22. Tiempo de respuesta pronosticado frente a tiempo de respuesta real.

[7] La línea inferior está basada en un ajuste de un polinomio de orden 3, a los datos disponibles para una carga de 0,5.

tante ver que el pronóstico parecía prometedor, pero que en realidad el sistema se viene abajo a partir de una carga de 0,8 o 0,9.

De esta forma, se necesita una herramienta de predicción más exacta. La opción 3 consiste en hacer uso de un modelo analítico, que puede ser expresado como un conjunto de ecuaciones que se pueden resolver para obtener los parámetros deseados (tiempo de respuesta, rendimiento, etc.). Para problemas de computadores, sistemas operativos y redes, e incluso para muchos problemas prácticos de la vida real, los modelos analíticos basados en la teoría de colas, proporcionan un buen ajuste a la realidad. La desventaja de la teoría de colas es que se deben realizar simplificaciones para derivar ecuaciones para los parámetros de interés.

El último enfoque es un modelo de simulación. Así, dado un lenguaje de programación de simulación lo suficientemente flexible y potente, el analista puede modelar la realidad en detalle y evitar muchas de las simplificaciones de la teoría de colas. Sin embargo, en la mayor parte de los casos, no se necesita un modelo de simulación o, al menos, no es aconsejable como un primer paso del análisis. Por una razón, tanto las medidas existentes como los pronósticos de carga futura conllevan un determinado margen de error. De forma que, sin importar lo bueno que es un modelo de simulación, el valor de los resultados está limitado por la calidad de las entradas. Por otra parte, a pesar de las suposiciones requeridas en la teoría de colas, los resultados obtenidos están muy cerca de los que se obtendrían con un análisis de simulación más cuidadoso. Además, un análisis de colas se puede realizar en minutos para problemas bien definidos, mientras que programar y ejecutar un ejercicio de simulación puede llevar días, semanas o incluso más tiempo.

Por consiguiente, es necesario que el analista domine los fundamentos de la teoría de colas.

COLAS DE UN SOLO SERVIDOR

La Figura 9.23 representa el sistema de colas más sencillo. La parte central del sistema es un servidor, que proporciona algún servicio a los elementos. Los elementos de una población de elementos llegan al sistema para ser atendidos. Si el servidor está ocioso, se atiende al elemento inmediatamente. De lo contrario, el elemento entrante se une a una línea de espera[8]. Cuando el servidor completa el servicio a un elemento, el elemento sale. Si hay elementos esperando en la cola, se manda uno inmediatamente al servidor. El servidor en este modelo puede representar cualquier cosa que realice alguna función o servicio a una colección de elementos. Ejemplos: un procesador da servicio a los procesos; una línea de transmisión proporciona un servicio de transmisión a los paquetes o estructuras de datos; un dispositivo de E/S proporciona un servicio de lectura o escritura a las peticiones de E/S.

La Tabla 9.8 resume algunos parámetros importantes asociados con un modelo de colas. Los elementos llegan con una determinada tasa media (elementos que llegan por segundo) λ. En un momento dado, un determinado número de elementos estará esperando en la cola (cero o más); el número medio esperando es ω, y el tiempo medio que cada elemento debe esperar es T_ω. T_ω se pondera sobre todos los elementos entrantes, incluyendo aquellos que no tienen que esperar. El servidor manipula los elementos entrantes con un tiempo de servicio medio T_s; este es el intervalo de tiempo entre el envío de un elemento al servidor y la salida de ese elemento del servidor. La utilización, ρ, es la fracción de tiempo que el servidor está ocupado, medida sobre cierto intervalo de tiempo. Finalmente hay dos parámetros del sistema global. El número medio de elementos residentes en el siste-

[8] En la literatura a la línea de espera se le denomina normalmente cola; también es usual referirse al sistema entero como una cola. A no ser que se indique otra cosa, usaremos el término *cola* para referirnos a la línea de espera.

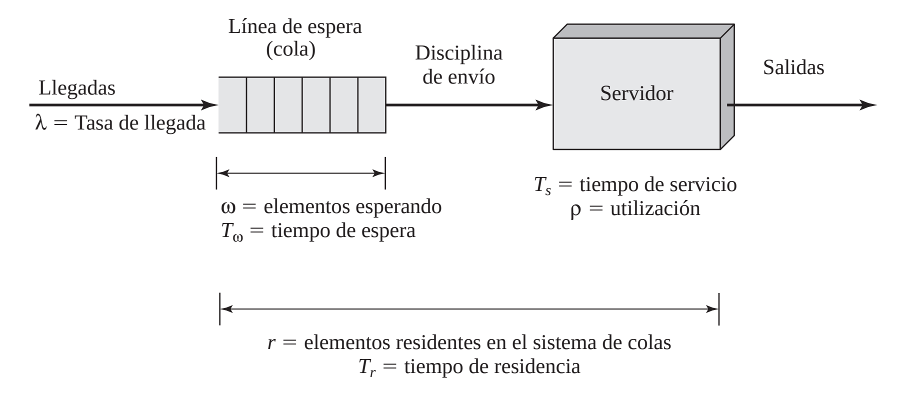


Figura 9.23. Estructura de un sistema de colas y parámetros para una cola con un solo servidor.

Tabla 9.8. Notación de los sistemas de colas.

λ	=	tasa de llegada; número medio de llegadas por segundo
T_s	=	tiempo medio de servicio para cada llegada; tiempo de servicio siendo atendido, sin contar el tiempo de espera en la cola
ρ	=	utilización; fracción de tiempo que las instalaciones (servidor o servidores) están ocupados
ω	=	número medio de elementos esperando a ser atendidos
T_ω	=	tiempo medio de espera (incluye los elementos que tienen que esperar y a los elementos con tiempo de espera = 0)
r	=	número medio de elementos residentes en el sistema (esperando o siendo atendidos)
T_r	=	tiempo medio de residencia; tiempo que un elemento pasa en el sistema (esperando o siendo atendido)

ma, incluyendo el elemento que está siendo atendido (si existe), es r; y el tiempo medio que un elemento está en el sistema, esperando y siendo atendido, es T_r; nos referiremos a él como el tiempo medio de residencia[9].

Si asumimos que la capacidad de una cola es infinita, no se pierde ningún elemento del sistema; simplemente se retrasan hasta que pueden ser atendidos. Bajo estas circunstancias, la tasa de salida se iguala a la tasa de entrada. A medida que se incrementa la tasa de entrada, la utilización se incrementa y, con ella, la congestión. La cola se vuelve mayor, incrementándose el tiempo de espera. Con $\rho = 1$, el servidor se satura, trabajando el 100% del tiempo. De esta forma, la tasa máxima de entrada teórica que puede ser gestionada por el sistema es

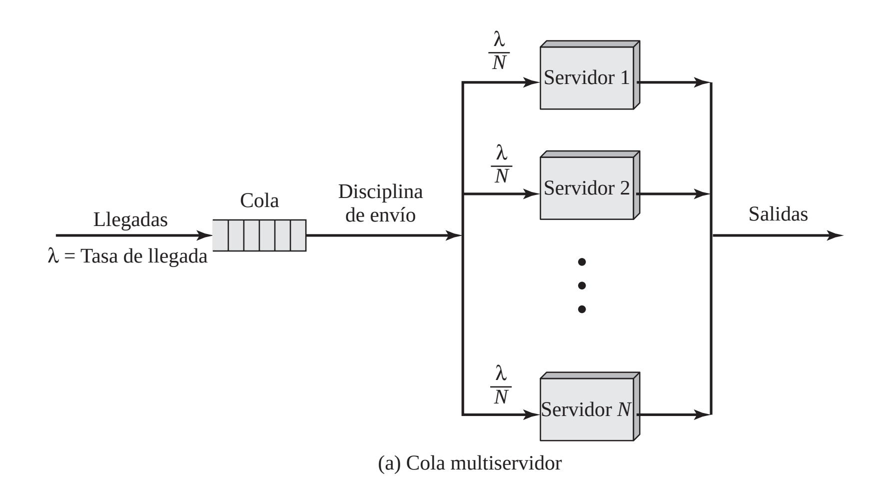
$$\lambda_{max} = \frac{1}{T_s}$$

[9] En parte de la literatura, a esta palabra se le denomina *tiempo medio en la cola*, mientras que otras veces se entiende el *tiempo medio en la cola* como el tiempo medio gastado esperando en la cola (antes de ser atendido).

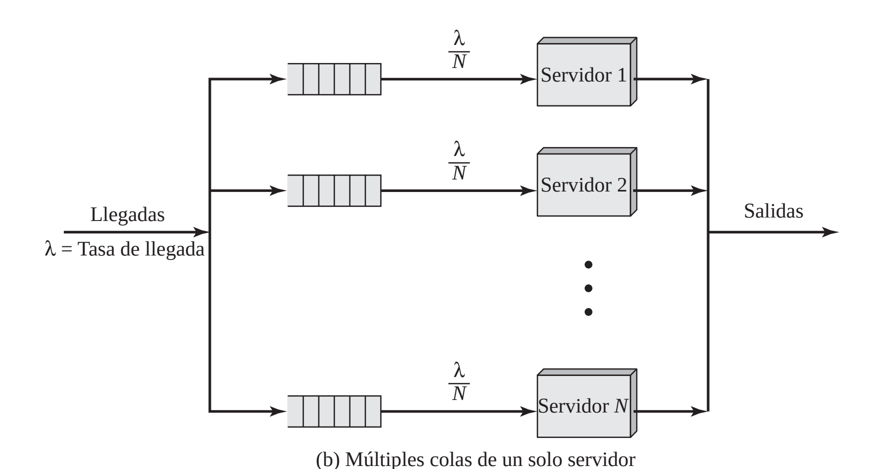
Sin embargo, las colas se hacen muy largas cuando se está saturando el sistema, creciendo sin límites cuando $\rho = 1$. Consideraciones prácticas, tales como los requisitos del tiempo de respuesta o el tamaño de los *buffers*, normalmente limitan la tasa de entrada para un solo servidor entre el 70 y 90% del máximo teórico.

Se suelen hacer las siguientes suposiciones:

- **Tamaño de la población.** De forma característica, se asume un tamaño infinito de la población. Esto significa que la tasa de llegada no se altera por la pérdida de población. Si la población es finita, entonces la población disponible de llegada se reduce por el número de elementos actualmente en el sistema; normalmente esto reduce la tasa de llegada proporcionalmente.
- **Tamaño de la cola.** Normalmente, se asume un tamaño de cola infinito. De esta forma, la línea de espera puede crecer sin límites. Con una cola finita, es posible que se pierdan elementos en el sistema. En la práctica, toda cola es finita. En la mayor parte de los casos, esto no cambia sustancialmente el análisis.



(a) Cola multiservidor



(b) Múltiples colas de un solo servidor

Figura 9.24. Cola multiservidor frente a múltiples colas de un solo servidor.

- **Disciplina de envío.** Cuando el servidor se queda libre, y si hay más de un elemento esperando, se debe decidir qué elemento enviar el siguiente. El enfoque más sencillo es primero-entra-primero-sale; esta disciplina es la que se usa normalmente cuando se utiliza el término *cola*. Otra posibilidad es último-entrar-primero-salir. Una cosa que se podría encontrar en la práctica es una disciplina de envío basada en el tiempo de servicio. Por ejemplo, un nodo de intercambio de paquetes podría decidir enviar los paquetes sobre la base de primero el más corto (para generar la mayor cantidad de paquetes salientes) o primero el más largo (para minimizar el tiempo de proceso relativo al tiempo de transmisión). Por desgracia, las disciplinas basadas en el tiempo de servicio son muy difíciles de modelar analíticamente.

LA COLA MULTISERVIDOR

La Figura 9.24 muestra una generalización del modelo simple para múltiples servidores, todos ellos compartiendo una cola común. Si llega un elemento y, al menos, hay un servidor disponible, se envía el elemento directamente a ese servidor. Se supone que todos los servidores son idénticos; es decir, si hay más de un servidor disponible, no hay diferencia sea cual sea el servidor que se elija para el elemento. Si todos los servidores están ocupados, se empieza a formar la cola. Tan pronto como un servidor se quede libre, se envía un elemento de la cola utilizando la disciplina de envío existente.

Con la excepción de la utilización, todos los parámetros mostrados en la Figura 9.23 se pueden utilizar en el caso multiservidor con la misma interpretación. Si tenemos N servidores idénticos, entonces ρ es la utilización de cada servidor y podemos considerar N_ρ como la utilización del sistema completo; este último término normalmente es conocido como intensidad de tráfico, u. De esta forma, la utilización máxima teórica es $N \times 100\%$, y la máxima tasa de entrada teórica es

$$\lambda_{\max} = \frac{N}{T_s}$$

Las características clave escogidas normalmente para una cola multiservidor son las mismas que para una cola de un solo servidor. Es decir, se supone población infinita y tamaño infinito de la cola, con una sola cola infinita compartida entre todos los servidores. A menos que se indique lo contrario, la disciplina de envío es FIFO. Para el caso multiservidor, si se suponen idénticos todos los servidores, la elección de un servidor particular para un elemento que está esperando, no tienen efecto sobre el tiempo de servicio.

Como contraste, la Figura 9.24b muestra la estructura de múltiples colas de un solo servidor.

PROYECTO DE PROGRAMACIÓN DOS. EL PLANIFICADOR DE HOST

El Sistema Operativo Hipotético de Pruebas (*Hypothetical Operating System Testbed*, HOST) es un sistema multiprogramado que tiene un planificador de procesos con cuatro niveles de prioridad, con la restricción de que los recursos disponibles son finitos.

Planificador con cuatro niveles de prioridad

El planificador opera con cuatro niveles de prioridad:

1. Procesos de tiempo real que deben ejecutar inmediatamente según la política primero en llegar primero en servirse (FCFS), expulsando a cualquier otro proceso que esté en ejecución y tenga menor prioridad. Estos procesos se ejecutan hasta que se completen.
2. Procesos normales de usuario que se ejecutan en un planificador retroalimentado de tres niveles (Figura P2.1). El *quantum* de tiempo del planificador es de 1 segundo. Éste también es el valor del *quantum* de tiempo de los planificadores retroalimentados.

El planificador necesita mantener dos colas de envío —prioridad de tiempo real y de usuario— que se alimentan desde la lista de trabajos que se deben planificar. Esta lista se examina cada *tick* del planificador y los trabajos que «han llegado» se transfieren a la cola adecuada. A continuación, se examinan las colas; cualquier trabajo de tiempo real se ejecuta hasta su finalización, expulsando a cualquier otro trabajo que esté ejecutando.

La cola de trabajos con prioridad de tiempo real debe estar vacía antes de que se reactive el planificador retroalimentado de menor prioridad. Cualquier trabajo con prioridad de usuario en la cola de trabajos de usuario que pueda ejecutar con los recursos disponibles (memoria y dispositivos de E/S) se transfiere a la cola de prioridad adecuada. El funcionamiento normal de la cola retroalimentada acepta todos los trabajos con el nivel máximo de prioridad y degrada su prioridad después de que se complete cada rodaja de tiempo. Sin embargo, el planificador puede aceptar trabajos con menor prioridad, en cuyo caso los inserta en la cola adecuada. Esta política le permite al planificador emular a un

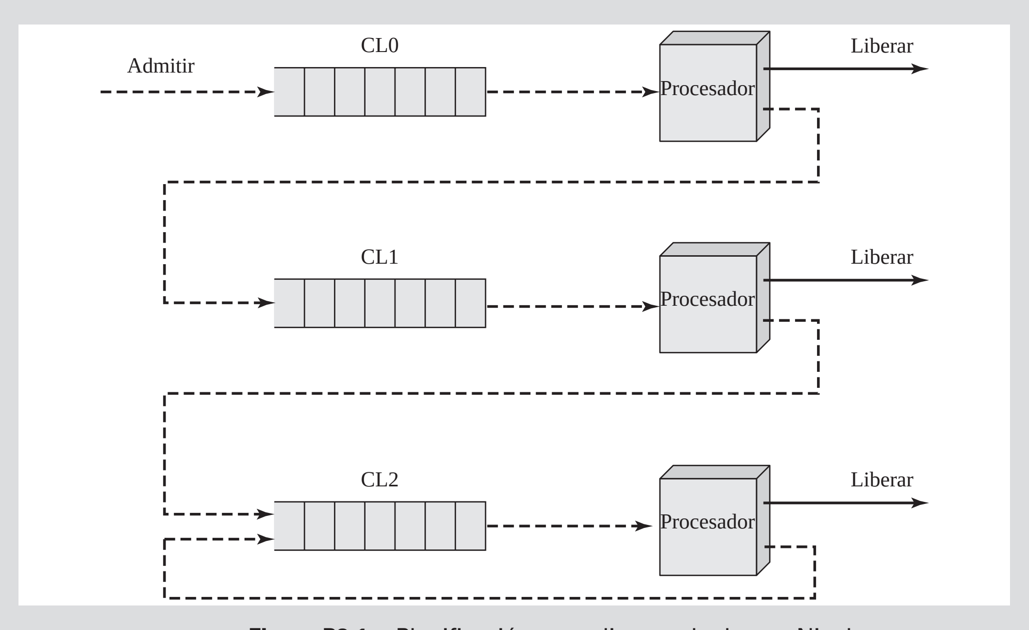


Figura P2.1. Planificación retroalimentada de tres Niveles.

planificador de tipo turno rotatorio (Figura P2.2) si se aceptan todos los trabajos con la menor prioridad posible.

Cuando se han finalizado todos los trabajos «listos» de mayor prioridad, el planificador retroalimentado se reanuda comenzando o continuando con el proceso que se encuentre al principio de la cola de mayor prioridad que no esté vacía. Al siguiente *tick* se suspende al trabajo actual (o se finaliza y se liberan sus recursos) si hay cualquier otro trabajo «listo» con una prioridad igual o mayor.

El flujo lógico deber ser el mostrado en la Figura P2.3 (y como el tratado en los ejercicios).

Restricciones de recursos

HOST tiene los siguientes recursos:

- 2 Impresoras
- 1 Escáner
- 1 Módem
- 2 Unidades de CD
- 1024 Mbytes de memoria disponible para los procesos

Los procesos con menor prioridad pueden utilizar alguno o todos los recursos, pero en el momento del envío del proceso se debe notificar al planificador HOST de los recursos que utilizará el proceso. El planificador asegura que cada recurso solicitado está solamente disponible para ese proceso a lo largo de todo su ciclo de vida en las colas: desde la transferencia inicial de la cola de trabajos a las colas de prioridad 1-3, hasta la finalización del proceso, incluyendo las rodajas de tiempo que no esté ejecutando.

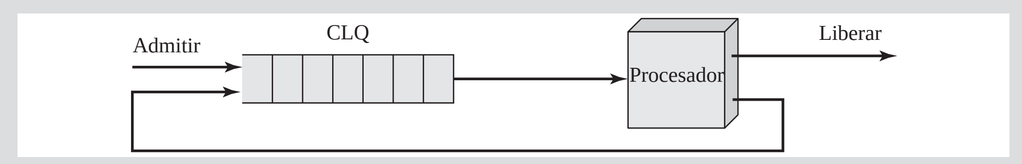


Figura P2.2. Planificador turno rotatorio.

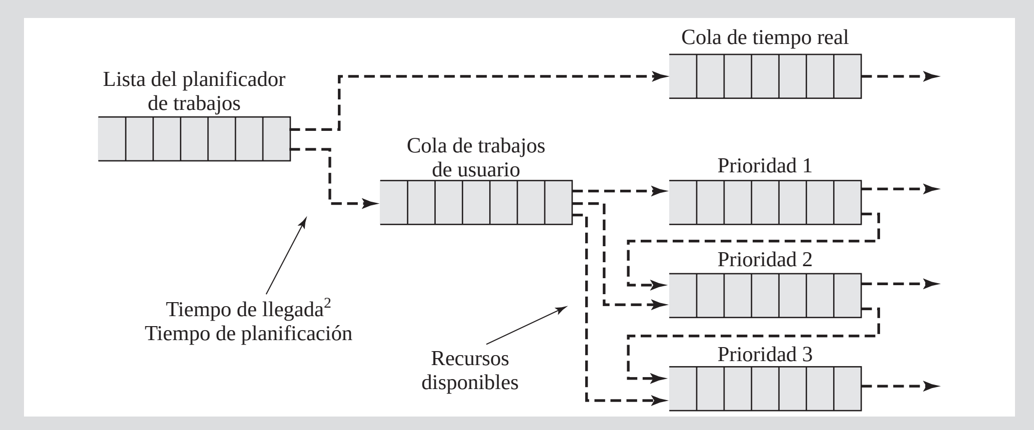


Figura P2.3. Flujo lógico del planificador.

Los procesos de tiempo real no necesitan ningún recurso de E/S (impresora / escáner / módem / CD), pero lógicamente necesitan asignación de memoria – para trabajos de tiempo real el requisito de memoria siempre será igual o menor a 64Mbytes.

Asignación de memoria

La asignación de memoria debe realizarse como un bloque contiguo de memoria para cada proceso. Esta asignación permanece asociada al proceso durante todo su tiempo de vida.

Se debe dejar suficiente memoria contigua libre para que los procesos de tiempo real no se queden bloqueados y puedan ejecutar – 64 Mbytes para cada trabajo de tiempo real en ejecución, dejando 960 Mbytes para ser compartidos entre los trabajos de usuario «activos».

La MMU del hardware de HOST no soporta memoria virtual, por lo que no es posible realizar intercambio entre memoria y disco. Tampoco es un sistema paginado.

Con estas restricciones, se puede utilizar cualquier esquema apropiado de asignación de memoria de particiones variables (Primer Encaje, Siguiente Encaje, Mejor Encaje, Peor Encaje, etc.).

Procesos

Los procesos en HOST son simulados por el planificador, creando un nuevo proceso para cada proceso que se debe planificar. Es un proceso genérico (suministrado como `proceso` – código fuente: `sigtrap.c`) que se puede ejecutar con cualquier prioridad. De hecho, se ejecuta a muy baja prioridad, durmiendo periodos de 1 segundo y mostrando:

1. Un mensaje con el ID del proceso, cuando comienza el proceso;
2. Un mensaje cada segundo que el proceso está en ejecución;
3. Un mensaje cuando se Suspende, Continúa o Finaliza el proceso.

El proceso finalizará por sí mismo a los 20 segundos, si no lo finaliza antes el planificador. La salida por pantalla del proceso se realiza con un color seleccionado al azar, por lo que las «rodajas» de los procesos se pueden distinguir fácilmente. Utilice este proceso mejor que uno propio.

El ciclo de vida de un proceso es:

1. Se envía el proceso a la cola de entrada del planificador a través de una lista de procesos inicial que define el tiempo de llegada, la prioridad, el tiempo de procesador requerido (en segundos), el tamaño del bloque de memoria y otros recursos solicitados.
2. Un proceso está «listo para ejecutar» cuando ha «llegado» y todos los recursos solicitados están disponibles.
3. Todos los trabajos de tiempo real pendientes se envían para su ejecución según la política primero en llegar primero en servirse (FCFS).
4. Si hay recursos disponibles suficientes y memoria para un proceso de usuario de menor prioridad, se transfiere el proceso a la cola de prioridad adecuada y se actualizan los indicadores de recursos restantes (lista de memoria y dispositivos de E/S).
5. Cuando se comienza un trabajo (`fork` y `exec("proceso", . . .)`), el planificador mostrará los parámetros del trabajo (ID de proceso, prioridad, tiempo restante de procesador (en segundos), posición de memoria y tamaño de bloque y recursos solicitados) antes de realizar el `exec`.
6. Se permite ejecutar a un proceso de tiempo real hasta que finalice, momento en el cual el planificador lo mata mandándole una señal `SIGINT`.
7. Se permite ejecutar a un trabajo de usuario de baja prioridad durante un *tick* del planificador (un segundo), momento tras el cual se suspende (`SIGTSTP`) o finaliza (`SIGINT`) si su tiempo

ha finalizado. Si se suspende, se baja su prioridad (si es posible) y se sitúa en la cola adecuada a esa prioridad, como se muestra en las Figuras P2.1 y P2.3. Para mantener la sincronización de la salida entre el planificador y el proceso hijo, el planificador debe esperar a que el proceso responda a la señal `SIGTSTP` o `SIGINT` antes de continuar `(waitpid(p->pid, &status, WUNTRACED))`. Para conseguir el rendimiento mostrado de las políticas de planificación en la Figura 9.5, el trabajo de usuario no se debe suspender y mover a una menor prioridad a menos que haya otro proceso esperando a ser (re)iniciado.

8. Siempre que no estén pendientes trabajos de tiempo real de mayor prioridad, se inicia o reinicia el proceso pendiente de mayor prioridad en las colas retroalimentadas `(SIGCONT)`.
9. Cuando se finaliza un proceso, los recursos que utilizó se devuelven al planificador para su asignación a otros recursos.
10. Cuando no hay más procesos en la lista de entrada, la cola de entrada y las colas retroalimentadas, finaliza el planificador.

Lista de planificación

La Lista de planificación es la lista de procesos que debe procesar el planificador.

La lista reside en un fichero de texto que se especifica en línea de mandatos, por ejemplo,

>hostd listaplanif

Cada línea de la lista describe un proceso con los siguientes datos separados por comas:

<tiempo de llegada>, <prioridad>, <tiempo de procesador>, <Mbytes>,

<#impresoras>, <#escáneres>, <#módems>, <#CDs>

De esta forma,

12, 0, 1, 64, 0, 0, 0, 0

12, 1, 2, 128, 1, 0, 0, 1

13, 3, 6, 128, 1, 0, 1, 2

indicaría:

Primer trabajo. Tiempo de llegada 12, prioridad 0 (tiempo real), requiere 1 segundo de tiempo de procesador y 64 Mbytes de memoria – no requiere recursos de E/S.

Segundo trabajo. Tiempo de llegada 12, prioridad 1 (máxima prioridad de trabajo de usuario), requiere 2 segundos de tiempo de procesador, 128 Mbytes de memoria, una impresora y 1 unidad de CD.

Tercer trabajo. Tiempo de llegada 13, prioridad 3 (mínima prioridad de trabajo de usuario), requiere 6 segundos de tiempo de procesador, 128 Mbytes de memoria, una impresora, un módem y 2 unidades de CD.

Este fichero de texto puede ser de cualquier longitud, con un contenido máximo de 1000 trabajos. Finalizará con una marca *end-of-line* (final-de-línea) seguida de *end-of-file* (final-de-fichero).

En los ejercicios se describen las listas de entrada al planificador para comprobar el funcionamiento de sus características. Estas listas son muy similares a las pruebas que tendrá que pasar el planificador durante su evaluación. Se espera un funcionamiento exactamente como el descrito en los ejercicios.

¡Obviamente, el planificador también será verificado con combinaciones más complejas!

Durante el curso se dispondrá de un planificador totalmente funcional. En caso de cualquier duda con la forma de funcionar del planificador o con el formato de salida, se puede utilizar este programa para verificar cómo se espera que funcione su planificador.

Requisitos del proyecto

1. Diseñe un planificador que satisfaga los criterios antes mencionados. En un documento formal de diseño:

   a) Describa y justifique qué algoritmos de asignación de memoria se podrían haber utilizado y justifique su decisión final.

   b) Describa y justifique las estructuras utilizadas por el planificador para las colas, planificación, asignación de memoria y otros recursos.

   c) Describa y justifique la estructura general de su programa, comentando sus módulos y principales funciones (se espera una descripción de las interfaces de las funciones).

   d) Justifique por qué se podría utilizar este esquema de planificación multinivel, comparándolo con esquemas utilizados por sistemas operativos «reales». Resuma las desventajas de este esquema y sugiera posibles soluciones. Incluya en su justificación los esquemas de asignación de memoria y recursos.

   Se espera que este documento formal tenga justificaciones, descripciones y argumentaciones exhaustivas. El documento de diseño se debe entregar impreso. El documento de diseño NO debe incluir código fuente.

2. Implemente el planificador utilizando el lenguaje C.
3. El código fuente **DEBE** estar extensamente comentado y apropiadamente estructurado, permitiendo a sus colegas comprenderlo y darle mantenimiento. ¡El código comentado con propiedad y bien alineado es mucho más fácil de interpretar e interesa que la persona que pueda evaluar su código pueda entenderlo con facilidad sin necesidad de hacer gimnasia mental!
4. Los detalles sobre el envío del proyecto se proporcionarán con antelación a la fecha límite.
5. El envío del proyecto debe contener sólo ficheros fuente, incluyendo ficheros de cabecera, y un `makefile`. No se debe incluir ningún fichero ejecutable. El evaluador recompilará automáticamente su intérprete de mandatos a partir del código fuente. Si el código fuente no compila, no será calificado.
6. El `makefile` (en letras minúsculas, por favor) **DEBE** generar un fichero binario llamado `hostd` (en letras minúsculas, por favor). Un ejemplo de `makefile` sería:

   # José Pérez, s1234567 – Sistemas Operativos – Proyecto 2

   # LabComp 1/01 tutor: Antonio López

   hostd: hostd.c utility.c hostd.h

   gcc hostd.c utility.c -o hostd

   El programa `hostd` se generará simplemente tecleando make en la línea de mandatos.

   Nota: la cuarta línea del makefile de ejemplo **DEBE** comenzar con un tabulador.

Documentación solicitada

1. Fichero(s) de código fuente, fichero(s) de cabecera y un `makefile`.
2. El documento de diseño descrito anteriormente.

Envío del código

Es necesario un `makefile`. Todos los ficheros incluidos en su envío se copiarán al mismo directorio, por tanto no incluya rutas en su `makefile`. El `makefile` debe incluir todas las dependencias para compilar el programa. Si se incluye una biblioteca, su `makefile` debe construir dicha biblioteca.

Para dejar esto claro: ***no construya a mano ningún binario o fichero objeto***. Todo lo que se requerirá serán sus ficheros fuente, un `makefile` y el fichero `readme`. Verifique su proyecto, copiando dichos ficheros a un directorio vacío y compilándolo completamente por medio del mandato `make`.

El corrector usará un *script* que copia sus ficheros a un directorio de prueba, borra el fichero `hostd` previo, así como todos los ficheros `*.a`, y/o los `*.o`, y ejecuta un `make`, copia una serie de ficheros de pruebas al directorio, y comprueba su planificador por medio de una serie de ficheros de prueba. Si esta batería de pruebas falla debido a nombres erróneos, diferencias entre mayúsculas y minúsculas, versiones erróneas de código que fallen al compilar o falta de ficheros, etc., la secuencia de evaluación se detendrá. En este caso, la calificación obtenida será la de las pruebas pasadas completamente, la puntuación sobre el código fuente y el manual.

CAPÍTULO 10

Planificación multiprocesador y de tiempo real

◆ ◆

Este capítulo continúa pasando revista a la planificación de procesos e hilos. Comenzamos examinando los aspectos que surgen de la disponibilidad de más de un procesador. Se exploran cierto número de decisiones de diseño. A continuación se verá la planificación de procesos en un sistema multiprocesador. Luego se examinan las consideraciones de diseño, un tanto diferentes, de la planificación de hilos en un multiprocesador. La segunda sección de este capítulo cubre la planificación de tiempo real. Esta sección comienza exponiendo las características de los procesos de tiempo real y luego analiza la naturaleza del proceso de planificación. Se examinan dos propuestas para la planificación de tiempo real, la planificación basada en plazos y la planificación de tasa monótona.

◆ ◆

10.1. PLANIFICACIÓN MULTIPROCESADOR

Cuando un sistema computador contiene más de un procesador, el diseño de la función de planificación plantea varias cuestiones nuevas. Se comienza con un breve resumen de los multiprocesadores y luego se ve que las consideraciones son bastante diferentes cuando la planificación se hace a nivel de proceso y cuando se hace nivel de hilo.

Podemos clasificar los sistemas multiprocesador como sigue:

- **Débilmente acoplado o multiprocesador distribuido, o cluster.** Consiste en una colección de sistemas relativamente autónomos; cada procesador tiene su propia memoria principal y canales de E/S. Tratamos este tipo de configuración en el Capítulo 14.
- **Procesadores de funcionalidad especializada.** Un ejemplo es un procesador de E/S. En este caso, hay un procesador de propósito general maestro y procesadores especializados que son controlados por el procesador maestro y que le proporcionan servicios. En el Capítulo 11 se tratan los aspectos relacionados con los procesadores de E/S.
- **Procesamiento fuertemente acoplado.** Consiste en un conjunto de procesadores que comparten la memoria principal y están bajo el control integrado de un único sistema operativo.

Lo que nos concierne en esta sección es la última categoría y específicamente los aspectos relacionados con la planificación.

GRANULARIDAD

Una buena manera de caracterizar los multiprocesadores y de situarlos en contexto respecto de otras arquitecturas, es considerar la granularidad de sincronización, o frecuencia de sincronización entre los procesos del sistema. Podemos distinguir cinco categorías de paralelismo que difieren en el grado de granularidad. Éstas se resumen en la Tabla 10.1, que está adaptada de [GEHR87] y [WOOD89].

Paralelismo Independiente Con paralelismo independiente, no hay sincronización explícita entre procesos. Cada uno representa un trabajo o aplicación independiente y separada. Un uso típico de este tipo de paralelismo se da en los sistemas de tiempo compartido. Cada usuario desarrolla su propio trabajo con una aplicación particular, como tratamiento de textos o una hoja de cálculo. El multiprocesador proporciona el mismo servicio que un monoprocesador multiprogramado. Dado que hay más de un procesador disponible, el tiempo de respuesta medio de los usuarios será menor.

Es posible conseguir una mejora de prestaciones similar proporcionando a cada usuario un computador personal o estación de trabajo. Si se va a compartir cualquier información o archivo, los siste-

mas individuales deben de estar conectados entre sí a través de una red, formando un sistema distribuido. Este enfoque se examina en el Capítulo 14. Por otro lado, un único sistema multiprocesador compartido es, en muchos casos, más eficaz en coste que un sistema distribuido, si consideramos la economía de escala en discos y otros periféricos.

Tabla 10.1. Granularidad de sincronización y procesos.

Tamaño del Grano	Descripción	Intervalo de sincronización (Instrucciones)
Fino	Paralelismo inherente en un único flujo de instrucciones	<20
Medio	Procesamiento paralelo o multitarea dentro de una única aplicación	20-200
Grueso	Multiprocesamiento de procesos concurrentes en un entorno multiprogramado	200-2000
Muy grueso	Procesamiento distribuido entre nodos de una red para conformar un único entorno de computación	2000-1M
Independiente	Múltiples procesos no relacionados	(N/D)

Paralelismo de grano grueso y muy grueso Con el paralelismo de grano grueso y muy grueso, hay sincronización entre procesos, pero a un nivel muy burdo. Este tipo de situación se trata sencillamente como un conjunto de procesos concurrentes ejecutando en un monoprocesador multiprogramado y puede proporcionarse en un multiprocesador con poco o ningún cambio en el software de usuario.

En [WOOD89] se da un ejemplo sencillo de una aplicación que puede explotar la existencia de un multiprocesador. Los autores han desarrollado un programa que, dada una especificación de archivos que necesitan ser recompilados para reconstruir un fragmento de software, determina cuál de estas compilaciones (normalmente todas ellas) pueden ser ejecutadas simultáneamente. El programa lanza un proceso por cada compilación paralela. Los autores informan que el aumento de velocidad en un multiprocesador realmente excede lo que cabría esperar si simplemente sumamos el número de procesadores en uso. Esto se debe a sinergias en las *caches* de disco (este tema se analiza en el Capítulo 11) y a la compartición del código del compilador, que está cargado en memoria una única vez.

En general, cualquier colección de procesos concurrentes que necesitan comunicarse o sincronizarse pueden beneficiarse del uso de una arquitectura multiprocesador. En el caso de que la interacción entre procesos sea muy poco frecuente, un sistema distribuido puede proporcionar un buen soporte. Sin embargo, si la interacción es algo más frecuente, entonces la sobrecarga de comunicaciones a través de la red puede mermar la potencial mejora de velocidad. En este caso, la organización multiprocesador proporciona un soporte más eficaz.

Paralelismo de grano medio Vimos en el Capítulo 4 que una aplicación individual puede ser implementada eficazmente como una colección de hilos dentro de un único proceso. En un caso como éste, el paralelismo potencial de la aplicación debe ser especificado explícitamente por el programador. De este modo, deberá haber un grado bastante alto de coordinación e interacción entre los hilos de la aplicación, dando lugar a un nivel de sincronización de grano medio.

Mientras que los paralelismos independiente, grueso y muy grueso pueden proporcionarse en un monoprocesador multiprogramado o en un multiprocesador con poco o ningún impacto en la función

de planificación, cuando se trata de la planificación de hilos es necesario reexaminar la planificación. Dado que varios hilos de una aplicación interactúan muy frecuentemente, las decisiones de planificación concernientes a un hilo pueden afectar a las prestaciones de la aplicación completa. Volveremos sobre este tema más adelante dentro de esta sección.

Paralelismo de grano fino El paralelismo de grano fino representa un uso mucho más complejo del paralelismo del que se encuentra en el uso de hilos. Aunque se ha realizado mucho trabajo sobre aplicaciones altamente paralelas, esta es, a día de hoy, un área especializada y fragmentada con muchas propuestas diferentes.

ASPECTOS DE DISEÑO

En un multiprocesador la planificación involucra tres aspectos interrelacionados:

- La asignación de procesos a procesadores.
- El uso de la multiprogramación en cada procesador individual.
- La activación del proceso, propiamente dicha.

Al considerar estos tres aspectos, es importante tener en mente que la propuesta elegida dependerá, en general, del grado de granularidad de la aplicación y del número de procesadores disponibles.

Asignación de procesos a procesadores Si se asume que la arquitectura del multiprocesador es uniforme, en el sentido de que ningún procesador tiene una ventaja física particular con respecto al acceso a memoria principal o a dispositivos de E/S, entonces el enfoque más simple de la planificación consiste en tratar cada proceso como un recurso colectivo y asignar procesos a procesadores por demanda. Surge la cuestión de si la asignación debería ser estática o dinámica.

Si un proceso se vincula permanentemente a un procesador desde su activación hasta que concluye, entonces se mantiene una cola a corto plazo dedicada por cada procesador. Una ventaja de esta estrategia es que puede haber menos sobrecarga en la función de planificación, dado que la asignación a un procesador se realiza una vez y para siempre. Asimismo, el uso de procesadores dedicados permite una estrategia conocida como planificación de grupo o pandilla, como se verá más adelante.

Una desventaja de la asignación estática es que un procesador puede estar ocioso, con su cola vacía, mientras otro procesador tiene trabajo acumulado. Para evitar esta situación, puede utilizarse una cola común. Todos los procesos van a una cola global y son planificados sobre cualquier procesador disponible. Así, a lo largo de la vida de un proceso, puede ser ejecutado en diferentes procesadores en diferentes momentos. En una arquitectura de memoria compartida fuertemente acoplada, la información de contexto de todos los procesos estará disponible para todos los procesadores, y por lo tanto el coste de planificación de un proceso será independiente de la identidad del procesador sobre cual se planifica. Otra opción más, es el balance dinámico de carga, en el que los hilos se mueven de una cola de un procesador a la cola de otro procesador; Linux utiliza este enfoque.

Independientemente de cómo los procesos se vinculan a los procesadores, se necesita alguna manera de asignar procesos a procesadores. Se han venido usando dos enfoques: maestro/esclavo y camaradas. Con la arquitectura maestro/esclavo, ciertas funciones clave del núcleo del sistema operativo ejecutan siempre en un procesador concreto. Los otros procesadores sólo pueden ejecutar programas de usuario. El maestro es responsable de la planificación de trabajos. Una vez que el pro-

ceso está activo, si el esclavo necesita servicio (por ejemplo, una llamada de E/S), debe enviar una solicitud al maestro y esperar a que el servicio se realice. Este enfoque es bastante simple y sólo requiere mejorar un poco un sistema operativo monoprocesador multiprogramado. La resolución de conflictos se simplifica porque un procesador tiene todo el control de la memoria y de los recursos de E/S. Hay dos desventajas en este enfoque: (1) un fallo en el maestro hace que falle el sistema completo, y (2) el maestro puede llegar a ser un cuello de botella para el rendimiento del sistema.

En la arquitectura camaradas, el núcleo puede ser ejecutado en cualquier procesador, y cada procesador se auto-planifica desde la colección de procesos disponibles. Este enfoque complica el sistema operativo. El sistema operativo debe asegurar que dos procesadores no escogen el mismo proceso y que los procesos no se extravían por ninguna razón. Deben emplearse técnicas para resolver y sincronizar la competencia en la demanda de recursos.

Por supuesto, hay una gama de opciones entre estos dos extremos. Un enfoque es tener un subconjunto de los procesadores dedicado a procesar el núcleo en vez de uno solo. Otro enfoque es simplemente gestionar las diferentes necesidades entre procesos del núcleo y otros procesos sobre la base de la prioridad y la historia de ejecución.

El uso de la multiprogramación en procesadores individuales Cuando cada proceso se asocia estáticamente a un procesador para todo su tiempo de vida, surge una nueva pregunta: ¿debería ese procesador ser multiprogramado? La primera reacción del lector puede ser preguntarse el porqué de esta cuestión; parece particularmente ineficiente vincular un procesador con un único proceso cuando este proceso puede quedar frecuentemente bloqueado esperando por E/S o por otras consideraciones de concurrencia/sincronización.

En los multiprocesadores tradicionales, que tratan con sincronizaciones de grano grueso o independientes (*véase* la Tabla 10.1), está claro que cada procesador individual debe ser capaz de cambiar entre varios procesos para conseguir una alta utilización y por tanto un mejor rendimiento. No obstante, para aplicaciones de grano medio ejecutando en un multiprocesador con muchos procesadores, la situación está menos clara. Cuando hay muchos procesadores disponibles, conseguir que cada procesador esté ocupado tanto tiempo como sea posible deja de ser lo más importante. En cambio, la preocupación es proporcionar el mejor rendimiento, medio, de las aplicaciones. Una aplicación que consista en varios hilos, puede ejecutar con dificultad a menos que todos sus hilos estén dispuestos a ejecutar simultáneamente.

Activación de procesos El último aspecto de diseño relacionado con la planificación multiprocesador, es la elección real del proceso a ejecutar. Hemos visto que en un monoprocesador multiprogramado, el uso de prioridades o de sofisticados algoritmos de planificación basados en el uso pasado, pueden mejorar el rendimiento frente a la ingenua estrategia FCFS (primero en llegar, primero en serservido). Cuando consideramos multiprocesadores, estas complejidades pueden ser innecesarias o incluso contraproducentes, y un enfoque más simple puede ser más eficaz con menos sobrecarga. En el caso de la planificación de hilos, entran en juego nuevos aspectos que pueden ser más importantes que las prioridades o las historias de ejecución. Tratemos cada uno de estos temas por turno.

PLANIFICACIÓN DE PROCESOS

En los sistemas multiprocesador más tradicionales, los procesos no se vinculan a los procesadores. En cambio hay una única cola para todos los procesadores o, si se utiliza algún tipo de esquema basado en prioridades, hay múltiples colas basadas en prioridad, alimentando a un único colectivo de procesadores. En cualquier caso, el sistema puede verse como una arquitectura de colas multiservidor.

Considérese el caso de un sistema biprocesador en el que cada procesador tiene la mitad de velocidad de procesamiento que un procesador en un sistema monoprocesador. [SAUE81] informa sobre

un análisis de colas que compara de las planificaciones FCFS, turno circular y tiempo restante más breve. El estudio se ocupa del tiempo de servicio de los procesos, que mide la cantidad de tiempo de procesador que necesita un proceso, bien para el trabajo completo o cada vez que el proceso está listo para utilizar el procesador. En el caso del turno circular, se asume que el cuanto de tiempo es largo comparado con la sobrecarga de cambio de contexto y pequeño comparado con el tiempo medio de servicio. Los resultados dependen de la variabilidad que se observa en los tiempos de servicio. Una medida común de la variabilidad es el coeficiente de variación, C_s[1]. Un valor de $C_s = 0$ corresponde al caso donde no hay variabilidad: los tiempos de servicio de todos los procesos son iguales. Valores mayores de C_s corresponden a una mayor variabilidad entre los tiempos de servicio. Esto es, a mayor valor de C_s, en mayor medida varían los valores de tiempos de servicio. Valores de C_s de 5 o más no son inusuales en distribuciones del tiempo de servicio de procesador.

La Figura 10.1a compara la productividad del turno circular con el FCFS en función de C_s. Nótese que la diferencia entre los algoritmos de planificación es mucho menor en el caso del biprocesador. Con dos procesadores, un proceso único con un tiempo de servicio largo es mucho menos disruptivo en el caso FCFS; otros procesos pueden utilizar el otro procesador. Resultados similares se muestran en la Figura 10.1b.

El estudio en [SAUE81] repite este análisis bajo cierto número de supuestos acerca del grado de multiprogramación, mezcla de procesos limitados por E/S o CPU, y el uso de prioridades. La conclusión general es que la disciplina de planificación específica es mucho menos importante con dos procesadores que con uno. Debería ser evidente que esta conclusión es incluso más fuerte a medida que el número de procesadores crece. Así, la disciplina básica FCFS o el uso de FCFS con un esquema estático de prioridades puede ser suficiente en un sistema multiprocesador.

PLANIFICACIÓN DE HILOS

Como hemos visto, con los hilos, el concepto de ejecución se separa del resto de la definición de un proceso. Una aplicación puede ser implementada como un conjunto de hilos, que cooperan y ejecutan de forma concurrente en el mismo espacio de direcciones.

En un monoprocesador, los hilos pueden usarse como una ayuda a la estructuración de programas y para solapar E/S con el procesamiento. Dada la mínima penalización por realizar un cambio de hilo comparado con un cambio de proceso, los beneficios se obtienen con poco coste.

Sin embargo, el poder completo de los hilos se vuelve evidente en un sistema multiprocesador. En este entorno, los hilos pueden explotar paralelismo real dentro de una aplicación. Si los hilos de una aplicación están ejecutando simultáneamente en procesadores separados, es posible una mejora drástica de sus prestaciones. Sin embargo, puede demostrarse que para aplicaciones que necesitan una interacción significativa entre hilos (paralelismo de grano medio), pequeñas diferencias en la gestión y planificación de hilos pueden dar lugar a un impacto significativo en las prestaciones [ANDE89].

Entre las muchas propuestas para la planificación multiprocesador de hilos y la asignación a procesadores, destacan cuatro enfoques generales:

- **Compartición de carga.** Los procesos no se asignan a un procesador particular. Se mantiene una cola global de hilos listos, y cada procesador, cuando está ocioso, selecciona un hilo de la

[1] El valor C_s, se calcula como σ_s/T_s donde σ_s es la desviación estándar del tiempo de servicio y T_s es el tiempo medio de servicio. En el documento sobre análisis de colas en **WilliamStallings.com/StudentSupport.html** encontrará más explicaciones sobre C_s.

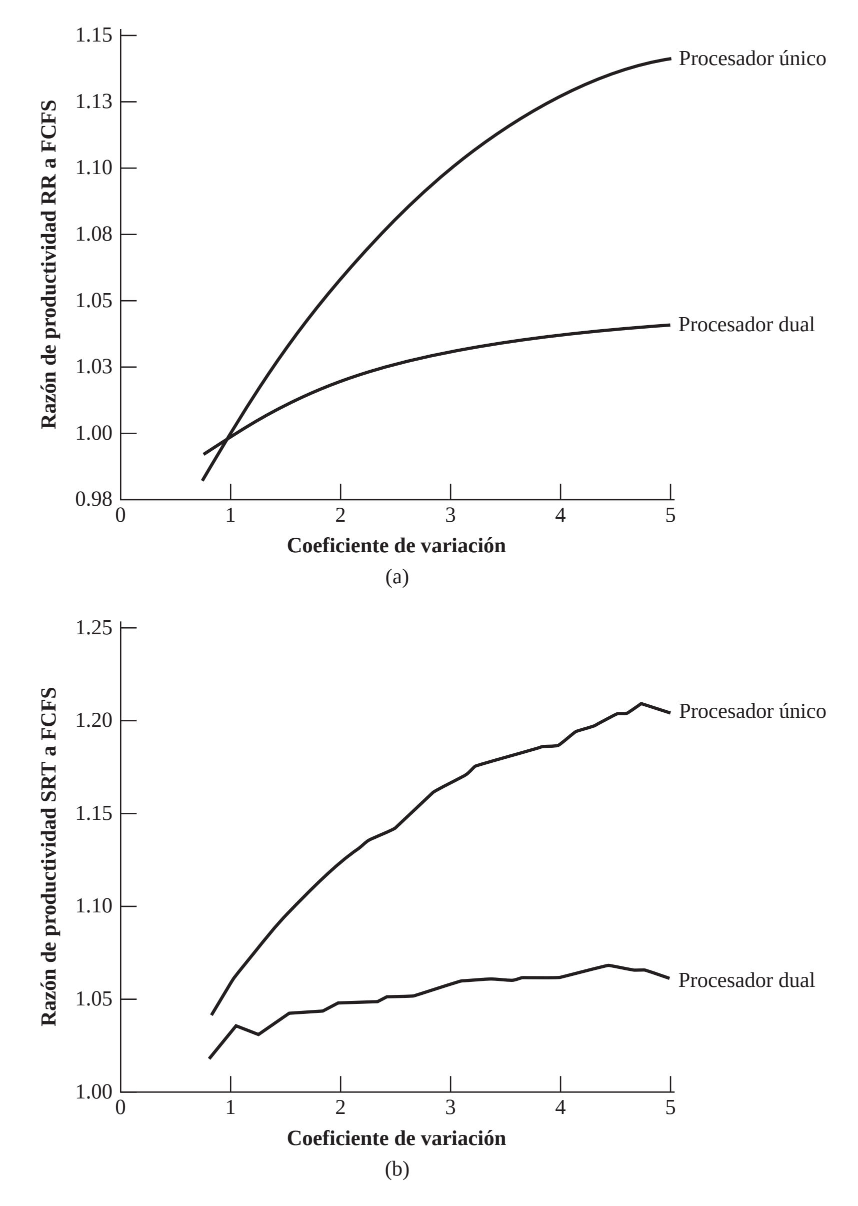


Figura 10.1. Comparación de rendimiento de planificación para uno y dos procesadores.

cola. El término **compartición de carga** se utiliza para distinguir esta estrategia de los esquemas de balanceo de carga en los que los trabajos se asignan de una manera más permanente (por ejemplo, *véase* [FEIT90a])[2].

---

[2] Cierta literatura sobre este tema se refiere a este enfoque como auto-planificación, porque cada procesador se planifica a sí mismo sin considerar a los otros procesadores. Sin embargo, este término también se utiliza en la literatura para referirse a programas escritos en lenguajes que permiten que el programador especifique la planificación (por ejemplo, *véase* [FOST91]).

- **Planificación en pandilla.** Un conjunto de hilos relacionados que se planifica para ejecutar sobre un conjunto de procesadores al mismo tiempo, en una relación uno-a-uno.
- **Asignación de procesador dedicado.** Esto es lo opuesto al enfoque de compartición de carga y proporciona una planificación implícita definida por la asignación de hilos a procesadores. Cada proceso ocupa un número de procesadores igual al número de hilos en el programa, durante toda la ejecución del programa. Cuando el programa termina, los procesadores regresan al parque general para la posible asignación a otro programa.
- **Planificación dinámica.** El número de hilos de un proceso puede cambiar durante el curso de su ejecución.

Compartición de carga La compartición de carga es posiblemente el enfoque más simple y que se surge más directamente de un entorno monoprocesador. Tiene algunas ventajas:

- La carga se distribuye uniformemente entre los procesadores, asegurando que un procesador no queda ocioso mientras haya trabajo pendiente.
- No se precisa un planificador centralizado; cuando un procesador queda disponible, la rutina de planificación del sistema operativo se ejecuta en dicho procesador para seleccionar el siguiente hilo.
- La cola global puede organizarse y ser accesible usando cualquiera de los esquemas expuestos en el Capítulo 9, incluyendo esquemas basados en prioridad y esquemas que consideran la historia de ejecución o anticipan demandas de procesamiento.

[LEUT90] analiza tres versiones diferentes de compartición de carga:

- **Primero en llegar, primero en ser servido (FCFS).** Cuando llega un trabajo, cada uno de sus hilos se disponen consecutivamente al final de la cola compartida. Cuando un procesador pasa a estar ocioso, coge el siguiente hilo listo, que ejecuta hasta que se completa o se bloquea.
- **Menor número de hilos primero.** La cola compartida de listos se organiza como una cola de prioridad, con la mayor prioridad para los hilos de los trabajos con el menor número de hilos no planificados. Los trabajos con igual prioridad se ordenan de acuerdo con qué trabajo llega primero. Al igual que con FCFS, el hilo planificado ejecuta hasta que se completa o se bloquea.
- **Menor número de hilos primero con expulsión.** Se le da mayor prioridad a los trabajos con el menor número de hilos no planificados. Si llega un trabajo con menor número de hilos que un trabajo en ejecución se expulsarán los hilos pertenecientes al trabajo planificado.

Usando modelos de simulación, los autores informan que, sobre un amplio rango de características de trabajo, FCFS es superior a las otras dos políticas en la lista precedente. Es más, los autores muestran que cierta forma de planificación en pandilla, expuesta en la siguiente subsección, es generalmente superior a la compartición de carga.

La compartición de carga tiene varias desventajas:

- La cola central ocupa una región de memoria a la que debe accederse de manera que se cumpla la exclusión mutua. De manera que puede convertirse en un cuello de botella si muchos procesadores buscan trabajo al mismo tiempo. Cuando hay sólo un pequeño número de procesadores, es difícil que esto se convierta en un problema apreciable. Sin embargo, cuando el multiprocesador consiste en docenas o quizás cientos de procesadores, la posibilidad de que esto sea un cuello de botella es real.

- Es poco probable que los hilos expulsados retomen su ejecución en el mismo procesador. Si cada procesador está equipado con una *cache* local, ésta se volverá menos eficaz.
- Si todos los hilos se tratan como un conjunto común de hilos, es poco probable que todos los hilos de un programa ganen acceso a procesadores a la vez. Si se necesita un alto grado de coordinación entre los hilos de un programa, los cambios de proceso necesarios pueden comprometer seriamente el rendimiento.

A pesar de las potenciales desventajas, este es uno de los esquemas más utilizados en los multiprocesadores actuales.

El sistema operativo Mach [BLAC90, WEND98] utiliza una técnica mejorada de compartición de carga. El sistema operativo mantiene una cola local por cada procesador y una cola global compartida. La cola local se utiliza para hilos que han sido temporalmente vinculados a un procesador concreto. Cada procesador examina primero su cola local para dar preferencia absoluta a los hilos ya vinculados sobre los no vinculados todavía. Como un ejemplo del uso de hilos vinculados, uno o más procesadores podrían estar dedicados a ejecutar procesos que forman parte del sistema operativo. Otro ejemplo es que los hilos de una aplicación particular podrían estar distribuidos entre ciertos procesadores; con el software adicional adecuado, esto proporciona soporte para la planificación en pandilla, expuesta a continuación.

Planificación en pandilla El concepto de planificar un conjunto de procesos simultáneamente sobre un conjunto de procesadores es anterior al uso de hilos. [JONE80] se refiere al concepto de planificación en grupo y cita los siguientes beneficios:

- Si se ejecutan en paralelo procesos estrechamente relacionados, puede reducirse el bloqueo por sincronización, pueden necesitarse menos cambios de proceso y las prestaciones aumentarán.
- La sobrecarga de planificación puede reducirse dado que una decisión única afecta a varios procesadores y procesos a la vez.

En el multiprocesador Cm*, se utiliza el término *coplanificación* [GEHR87]. La coplanificación se basa en el concepto de planificar un conjunto de tareas relacionadas, denominadas carga de trabajo. Los elementos individuales de la carga de trabajo tienden a ser bastante pequeños y por tanto próximos a la idea de hilo.

El término planificación en pandilla se ha aplicado a la planificación simultánea de los hilos que componen un proceso individual [FEIT90b]. La planificación en pandilla es para aplicaciones paralelas de grano medio o fino cuyo rendimiento se ve degradado de forma importante cuando cualquier parte de la aplicación no está ejecutando mientras otras partes están listas para ejecutar. También es beneficiosa para cualquier aplicación paralela incluso para aquellas de rendimiento no tan sensible. La necesidad de planificación en pandilla está ampliamente reconocida, y existen implementaciones en variedad de sistemas operativos multiprocesador.

Una manera obvia en que la planificación en pandilla mejora las prestaciones de una aplicación individual es porque se minimizan los cambios de proceso. Suponga que un hilo de un proceso está ejecutando y alcanza un punto en el que debe sincronizarse con otro hilo del mismo proceso. Si este otro hilo no está ejecutando, pero está listo para ejecutar, el primer hilo quedará colgado hasta que suceda un cambio de proceso en otro procesador que tome el hilo necesario. En una aplicación con coordinación estrecha entre sus hilos, estos cambios reducirán dramáticamente el rendimiento. La planificación simultánea de hilos cooperantes puede también reducir el tiempo de ubicación de recursos. Por ejemplo, múltiples hilos planificados en pandilla pueden acceder a un fichero sin la sobrecarga adicional de bloquearse durante una operación de posicionamiento y lectura/escritura.

El uso de la planificación en pandilla crea un requisito para la ubicación de procesador. Una posibilidad es la siguiente: suponga que tenemos *N* procesadores y *M* aplicaciones, cada una de las cuales tiene *N* hilos o menos. Entonces a cada aplicación puede dársele 1/*M* del tiempo disponible en los *N* procesadores, usando porciones de tiempo. [FEIT90a] hace notar que esta estrategia puede ser ineficiente. Considere un ejemplo en el que hay dos aplicaciones, una con cuatro hilos y otra con un hilo. Utilizando una asignación de tiempo uniforme se desperdicia el 37,5% del recurso de procesamiento, dado que cuando ejecuta la aplicación de un único hilo, tres de los procesadores quedan ociosos (*véase* Figura 10.2). Si hay varias aplicaciones de un único hilo, todas ellas podrían ser encajadas juntas para mejorar la utilización de los procesadores. Si esa opción no está disponible, una alternativa a la planificación uniforme es la planificación ponderada por el número de hilos. Así, a la aplicación de cuatro hilos podría dársele 4/5 del tiempo y a cada aplicación de un hilo dársele solamente un quinto del tiempo, reduciéndose el desperdicio de procesador a un 15%.

Asignación de procesador dedicado Una forma extrema de la planificación en pandilla, sugerida en [TUCK89], es dedicar un grupo de procesadores a una aplicación durante toda la duración de la aplicación. Esto es, cuando se planifica una aplicación dada, cada uno de sus hilos se asigna a un procesador que permanece dedicado al hilo hasta que la aplicación concluye.

Este enfoque puede parecer un desperdicio extremo del tiempo de procesador. Si un hilo de una aplicación se bloquea esperando por E/S o por sincronización con otro hilo, entonces el procesador de ese hilo se queda ocioso: no hay multiprogramación de los procesadores. Pueden realizarse dos observaciones en defensa de esta estrategia:

1. En un sistema altamente paralelo, con decenas o cientos de procesadores, cada uno de los cuales representa una pequeña fracción del coste del sistema, la utilización del procesador deja de ser tan importante como medida de la eficacia o rendimiento.

2. Evitar totalmente el cambio de proceso durante la vida de un programa debe llevar a una sustancial mejora de velocidad de ese programa.

Tanto [TUCK89] como [ZAHO90] informan de los análisis que sustentan la segunda afirmación. La Figura 10.3 muestra los resultados de un experimento [TUCK89]. Los autores ejecutaron dos aplicaciones, una multiplicación de matrices y el cálculo de una transformada rápida de Fourier (FFT), en un sistema con 16 procesadores. Cada aplicación descompone su problema en varias tareas, que se proyectan en hilos que dicha aplicación ejecuta. Los programas fueron escritos de manera tal que se pueda variar el número de hilos que usan. En esencia, cada aplicación define e introduce en una cola un cierto número de tareas. La aplicación extrae las tareas de la cola y las proyecta sobre los hilos disponibles. Si hay menos hilos que tareas, las tareas restantes permanecen enconadas hasta que son

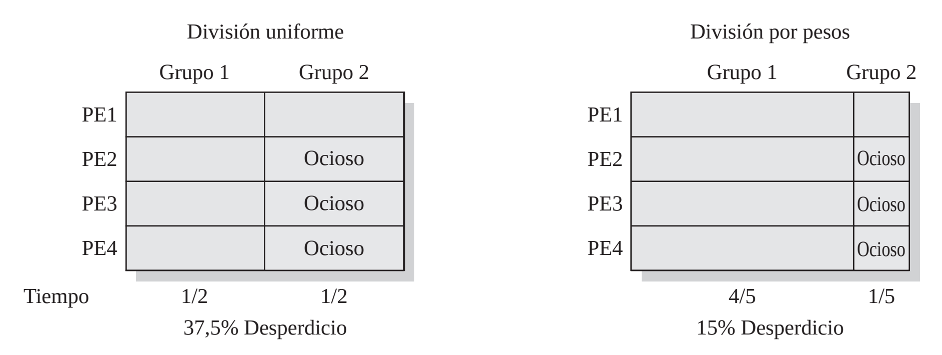


Figura 10.2. Ejemplo de planificación de grupos con cuatro y un hilos [FEIT90b].

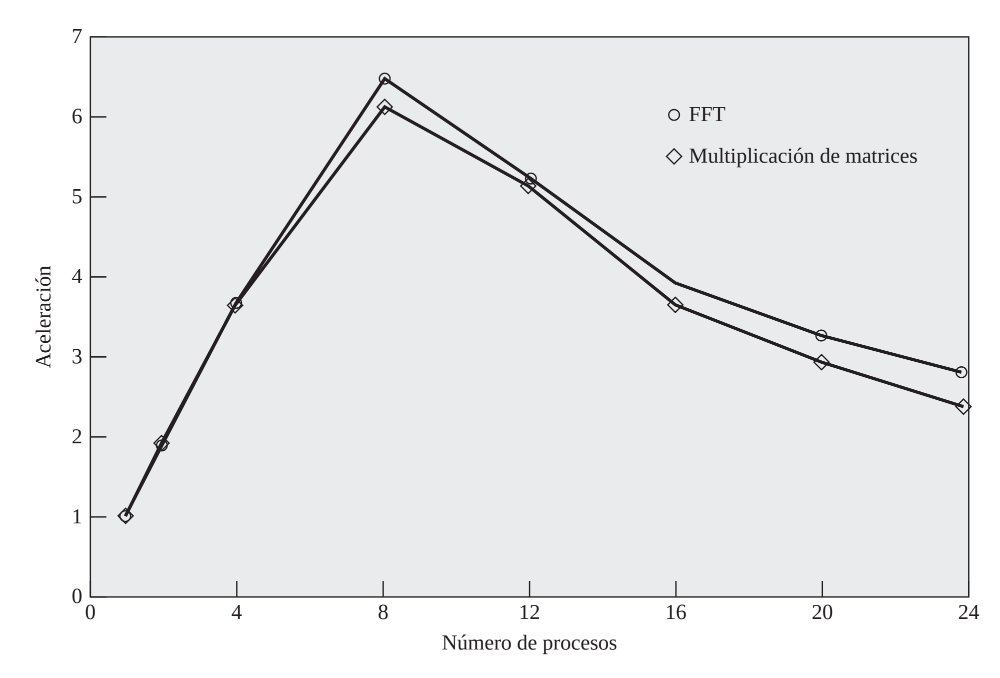


Figura 10.3. Aceleración de la aplicación en función del número de procesos [TUCK89].

extraídas por los hilos cuando completan la tarea asignada. Claramente, no todas las aplicaciones pueden ser estructuradas de este modo, pero muchos problemas numéricos y otras aplicaciones varias sí pueden tratarse de esta forma.

La Figura 10.3 muestra el incremento de velocidad de estas aplicaciones según el número de hilos que ejecutan las tareas de cada aplicación varía de 1 a 24. Por ejemplo, se observa que cuando ambas aplicaciones se arrancan simultáneamente con 24 hilos cada una, la aceleración obtenida, comparada con el uso de un único hilo por aplicación, es de 2,8 para la multiplicación de matrices y de 2,4 para la FFT. La figura muestra que el rendimiento de ambas aplicaciones empeora considerablemente cuando el número de hilos en cada aplicación excede 8 y, por tanto, el número total de procesos en el sistema excede el número de procesadores. Es más, a mayor número de hilos peor rendimiento se obtiene, dado que la frecuencia de expulsión de hilos y de re-planificación es mayor. Esta excesiva expulsión provoca ineficiencia en muchos sentidos: tiempo gastado esperando a que un hilo suspendido abandone una sección crítica, tiempo gastado en cambios de proceso e ineficiente comportamiento de la *cache*.

Los autores concluyen que una estrategia eficaz es limitar el número de hilos activos al número de procesadores en el sistema. Si la mayoría de las aplicaciones son de un hilo único o pueden ser estructuradas como una cola de tareas, se conseguirá un uso eficaz y razonablemente eficiente de los recursos procesador.

Tanto la asignación de procesador dedicado como la planificación en pandilla atacan el problema de la planificación tratando el aspecto de la ubicación del procesador. Puede observarse que el problema de la ubicación del procesador en un multiprocesador se parece más al problema de la ubicación de la memoria en un monoprocesador que al problema de la planificación en un monoprocesador. Ahora la pregunta es cuántos procesadores asignar a un programa en un momento dado, que es análoga a cuántos marcos de página asignar a un proceso en dicho momento. [GEHR87] define el término *conjunto residente de actividad*, análogo al de conjunto residente de la memoria virtual,

como el mínimo número de actividades (hilos) que deben ser planificados simultáneamente sobre procesadores para que la aplicación tenga un progreso aceptable. Al igual que con los esquemas de la gestión de la memoria, no planificar todos los elementos de un conjunto residente de actividad puede provocar trasiego de procesador. Esto sucede cuando la planificación de hilos cuyos servicios se necesitan, provoca la expulsión de otros hilos cuyos servicios serán necesitados pronto. De igual modo, la *fragmentación de procesador* se refiere a una situación en la cual quedan algunos procesadores sobrantes mientras otros están ubicados, y los procesadores sobrantes son o bien insuficientes en número, o bien no están organizados adecuadamente para dar soporte a los requisitos de las aplicaciones pendientes. La planificación en pandilla y la ubicación de procesador dedicado tienen como objetivo evitar estos problemas.

Planificación dinámica Para algunas aplicaciones, es posible proporcionar, en el lenguaje o en el sistema, herramientas que permitan que el número de hilos del proceso pueda ser alterado dinámicamente. Esto permitiría que el sistema operativo ajustase la carga para mejorar la utilización.

[ZAHO90] propone un enfoque en el cual tanto el sistema operativo como la aplicación están involucrados en tomar las decisiones de planificación. El sistema operativo es el responsable de particionar los procesadores entre los trabajos. Cada trabajo utiliza los procesadores actualmente en su partición para ejecutar algún subconjunto de sus tareas ejecutables proyectando estas tareas sobre hilos. La decisión apropiada acerca del subconjunto a ejecutar, así como qué hilos suspender cuando el proceso sea expulsado, se le deja a las aplicaciones individuales (quizás a través de un conjunto de rutinas de biblioteca). Este enfoque puede no ser adecuado para todas las aplicaciones. Sin embargo, algunas aplicaciones pueden ser consideradas como un único hilo mientras otras pueden ser programadas para sacar ventaja de esta particular característica del sistema operativo.

En este enfoque, la responsabilidad de planificación del sistema operativo está limitada fundamentalmente a la ubicación del procesador y se realiza de acuerdo a la siguiente política. Cuando un trabajo solicita uno o más procesadores (bien cuando el trabajo llega por primera vez o porque sus requisitos cambian),

1. Si hay procesadores ociosos, utilizarlos para satisfacer la solicitud.
2. En otro caso, si el trabajo que realiza la solicitud acaba de llegar, ubicarlo en un único procesador quitándoselo a cualquier trabajo que actualmente tenga más de un procesador.
3. Si no puede satisfacerse cualquier parte de la solicitud, mantenerla pendiente hasta que un procesador pase a estar disponible, o el trabajo rescinda la solicitud (por ejemplo, si dejan de ser necesarios los procesadores extra).

Cuando se libere uno o más procesadores (incluyendo la terminación de un trabajo),

4. Examinar la cola actual de solicitudes de procesador no satisfechas. Asignar un único procesador a cada trabajo en la lista que no tenga actualmente procesadores (por ejemplo, a todos los recién llegados en espera). Luego volver a examinar la lista, volviendo a asignar el resto de los procesadores siguiendo la estrategia FCFS.

[ZAHO90] y [MAJU88] informan sobre los análisis que sugieren que para las aplicaciones que pueden aprovechar las ventajas de la planificación dinámica, este enfoque es superior a la planificación en pandilla o a la asignación de procesador dedicado. No obstante, la sobrecarga de este enfoque puede invalidar esta aparente ventaja en rendimiento. Es necesario experimentar con el sistema real para probar la ventaja de la planificación dinámica.

10.2. PLANIFICACIÓN DE TIEMPO REAL

ANTECEDENTES

La computación de tiempo real se está convirtiendo en una disciplina de importancia cada vez mayor. El sistema operativo, y en particular el planificador, es quizás el componente más importante de un sistema de tiempo real. Ejemplos de aplicaciones actuales de sistemas de tiempo real son: control de experimentos de laboratorio, control de procesos en plantas industriales, robótica, control del tráfico aéreo, telecomunicaciones y sistemas militares de mando y control. La siguiente generación de sistemas incluirá vehículos todo terreno autónomos, controladores de robots con articulaciones elásticas, sistemas fundamentados en la manufactura inteligente, la estación espacial y la exploración del fondo marino.

La computación de tiempo real puede definirse como aquella en la que la corrección del sistema depende no sólo del resultado lógico de la computación sino también del momento en el que se producen los resultados. Podemos definir un sistema de tiempo real definiendo lo que se entiende por proceso de tiempo real, o tarea[3]. En general, en un sistema de tiempo real, algunas de las tareas son tareas de tiempo real, y éstas tienen cierto grado de urgencia. Tales tareas intentan controlar o reaccionar a eventos que tienen lugar en el mundo exterior. Dado que estos eventos ocurren en «tiempo real», una tarea de tiempo real debe ser capaz de mantener el ritmo de aquellos eventos que le conciernen. Así, normalmente es posible asociar un plazo de tiempo límite con una tarea concreta, donde tal plazo especifica el instante de comienzo o de finalización. Tal tarea puede ser clasificada como dura o blanda. Una **tarea de tiempo real duro** es aquella que debe cumplir su plazo límite; de otro modo se producirá un daño inaceptable o error fatal en el sistema. Una **tarea de tiempo real suave** tiene asociado un plazo límite deseable pero no obligatorio; sigue teniendo sentido planificar y completar la tarea incluso cuando su plazo límite ya haya vencido.

Otra característica de las tareas de tiempo real es si son periódicas o aperiódicas. Una **tarea aperiódica** tiene un plazo en el cual debe finalizar o comenzar, o puede tener una restricción tanto de su instante de comienzo como de finalización. En el caso de una **tarea periódica**, el requisito puede ser enunciado como «una vez por periodo *T*» o «exactamente *T* unidades a parte».

CARACTERÍSTICAS DE LOS SISTEMAS OPERATIVOS DE TIEMPO REAL

Los sistemas operativos de tiempo real pueden ser caracterizados por tener requisitos únicos en cinco áreas generales [MORG92]:

- Determinismo
- Reactividad
- Control del usuario
- Fiabilidad
- Operación de fallo suave

[3] Como suele pasar, la terminología presenta un problema, porque en la literatura se utilizan varias palabras con varios significados. Es común que un proceso particular opere bajo restricciones de tiempo real de naturaleza repetitiva. Esto es, el proceso dura un largo tiempo y, durante ese tiempo, realiza cierta función repetitiva en respuesta a eventos de tiempo-real. Durante esta sección, nos vamos a referir a una función individual como una tarea. Así, puede entenderse que el proceso avanza a través de una secuencia de tareas. En cualquier momento dado, el proceso está dedicando una única tarea y es ese proceso/tarea el que debe ser planificado.

Un sistema operativo es **determinista** en el sentido de que realiza operaciones en instantes de tiempo fijos predeterminados o dentro de intervalos de tiempo predeterminados. Cuando múltiples procesos compiten por recursos y tiempo de procesador, ningún sistema puede ser totalmente determinista. En un sistema operativo de tiempo real, las solicitudes de servicio de los procesos son dirigidas por eventos externos y temporizaciones. El grado en el que un sistema operativo puede satisfacer de manera determinista las solicitudes depende, primero, de la velocidad a la que es capaz de responder a las interrupciones y, segundo, de si el sistema tiene capacidad suficiente para manejar todas las solicitudes dentro del tiempo requerido.

Una medida útil de la capacidad de un sistema operativo de funcionar de manera determinista es el retardo máximo desde la llegada de una interrupción de un dispositivo de alta prioridad hasta que comienza el servicio. En sistemas operativos no de tiempo real, este retardo puede estar en el rango de decenas a cientos de milisegundos, mientras que en un sistema operativo de tiempo real este retardo puede tener un límite superior en algún punto entre los pocos microsegundos y el milisegundo.

Una característica distinta pero relacionada es la **reactividad**. El determinismo se preocupa de cuánto tiempo tarda el sistema operativo antes del reconocimiento de una interrupción. La reactividad se preocupa de cuánto tiempo tarda el sistema operativo, después del reconocimiento, en servir la interrupción. La reactividad incluye los siguientes aspectos:

1. La cantidad de tiempo necesario para manejar inicialmente la interrupción y comenzar a ejecutar la rutina de servicio de la interrupción (RSI). Si la ejecución de la RSI necesita un cambio de proceso, entonces el retardo será mayor que si la RSI puede ser ejecutada dentro del contexto del proceso actual.
2. La cantidad de tiempo necesario para realizar la RSI. Esto depende, generalmente, de la plataforma hardware.
3. El efecto del anidamiento de interrupciones. Si una RSI puede ser interrumpida por la llegada de otra interrupción, entonces el servicio se retrasará.

El determinismo y la reactividad juntos conforman el tiempo de respuesta a eventos externos. Los requisitos de tiempo de respuesta son críticos para los sistemas de tiempo real, dado que estos sistemas deben cumplir requisitos de tiempo impuestos por individuos, dispositivos o flujos de datos externos al sistema.

El **control del usuario** es generalmente mucho mayor en un sistema operativo de tiempo real que en sistemas operativos ordinarios. En un típico sistema operativo no de tiempo real, o bien el usuario no tiene control sobre la función de planificación o bien el sistema operativo sólo puede proporcionar una guía burda, como la agrupación de usuarios en más de una clase de prioridad. En un sistema de tiempo real, sin embargo, es esencial permitirle al usuario un control de grano fino sobre la prioridad de la tarea. El usuario debe ser capaz de distinguir entre tareas duras y suaves y de especificar prioridades relativas dentro de cada clase. Un sistema de tiempo real puede también permitirle al usuario especificar características como el uso de paginación en los procesos, qué procesos deben residir siempre en memoria principal, qué algoritmos de transferencia a disco deben utilizarse, qué derechos tienen los procesos de las varias bandas de prioridad, etcétera.

La **fiabilidad** es normalmente mucho más importante para los sistemas de tiempo real que para los que no lo son. Un fallo transitorio en un sistema no de tiempo real puede solventarse simplemente rearrancando el sistema. El fallo de un procesador en un sistema multiprocesador no de tiempo real puede dar lugar a un nivel de servicio degradado hasta que el procesador que falla sea reparado o sustituido. Pero un sistema de tiempo real ha de responder y controlar eventos en tiempo real. La pérdida o degradación de sus prestaciones puede tener consecuencias catastróficas: pérdidas económicas, daños en equipos importantes e incluso pérdida de vidas.

Como en otras áreas, la diferencia entre sistema operativo de tiempo real y no de tiempo real es una cuestión de grado. Incluso un sistema de tiempo real debe estar diseñado para responder a varios modos de fallo. La **operación de fallo suave** es una característica que se refiere a la habilidad del sistema de fallar de tal manera que se preserve tanta capacidad y datos como sea posible. Por ejemplo, cuando un típico sistema UNIX tradicional detecta corrupción de datos dentro del núcleo, emite un mensaje de fallo en la consola del sistema, vuelca los contenidos de la memoria al disco para un análisis posterior del fallo, y termina la ejecución del sistema. En cambio, un sistema de tiempo real intentará o bien corregir el problema o bien minimizar sus efectos continuando, en todo caso, en ejecución. Normalmente, el sistema notifica a un usuario o a un proceso de usuario, que debe intentar una acción correctiva, y luego continúa operando quizás con un nivel de servicio reducido. En el caso de que sea necesario apagar la máquina, se intentará mantener la consistencia de ficheros y datos.

Un aspecto importante de la operación de fallo suave se conoce como estabilidad. Un sistema de tiempo real es estable si, en los casos en los que sea imposible cumplir los plazos de todas las tareas, el sistema cumplirá los plazos de sus tareas más críticas, de más alta prioridad, aunque los plazos de algunas tareas menos críticas no se satisfagan.

Para cumplir los requisitos precedentes, los sistemas operativos de tiempo real incluyen de forma representativa las siguientes características [STAN89]:

- Cambio de proceso o hilo rápido.
- Pequeño tamaño (que está asociado con funcionalidades mínimas).
- Capacidad para responder rápidamente a interrupciones externas.
- Multitarea con herramientas para la comunicación entre procesos como semáforos, señales y eventos.
- Utilización de ficheros secuenciales especiales que pueden acumular datos a alta velocidad.
- Planificación expulsiva basada en prioridades.
- Minimización de los intervalos durante los cuales se deshabilitan las interrupciones.
- Primitivas para retardar tareas durante una cantidad dada de tiempo y para parar/retomar tareas.
- Alarmas y temporizaciones especiales.

El corazón de un sistema de tiempo real es el planificador de tareas a corto plazo. En el diseño de tal planificador, la equidad y la minimización del tiempo medio de respuesta no son lo más importante. Lo que es importante es que todas las tareas de tiempo real duro se completen (o comiencen) en su plazo y que tantas tareas de tiempo real suave como sea posible también se completen (o comiencen) en su plazo.

La mayoría de los sistemas operativos de tiempo real contemporáneos son incapaces de tratar directamente con plazos límite. En cambio, se diseñan para ser tan sensibles como sea posible a las tareas de tiempo real de manera que, cuando se aproxime un plazo de tiempo, la tarea pueda ser planificada rápidamente. Desde este punto de vista, las aplicaciones de tiempo real requieren típicamente tiempos de respuesta deterministas en el rango de varios milisegundos al submilisegundo y bajo un amplio conjunto de condiciones; las aplicaciones más avanzadas —en simuladores de aviones militares, por ejemplo— suelen tener restricciones en el rango de los 10 a los 100 µs [ATLA80].

La Figura 10.4 ilustra un abanico de posibilidades. En un planificador expulsivo que utiliza una simple planificación de turno circular, una tarea de tiempo real sería añadida a la cola de listos para

esperar su próxima rodaja de tiempo, como se ve en la Figura 10.4a. En este caso, el tiempo de planificación será generalmente inaceptable para aplicaciones de tiempo real. De modo alternativo, con un planificador no expulsivo, puede utilizarse un mecanismo de planificación por prioridad, dándole a las tareas de tiempo real mayor prioridad. En este caso, una tarea de tiempo real que esté lista será planificada tan pronto el proceso actual se bloquee o concluya (Figura 10.4b). Esto podría dar lugar a un retardo de varios segundos si una tarea lenta de baja prioridad estuviera ejecutando en un momento crítico. Una vez más, este enfoque no es aceptable. Un enfoque más prometedor es combinar prioridades con interrupciones basadas en el reloj. Los puntos de expulsión ocurrirán a intervalos regulares. Cuando suceda un punto de expulsión, la tarea actualmente en ejecución será expulsada si hay esperando una tarea de mayor prioridad. Esto podría incluir la expulsión de tareas que son parte del núcleo del sistema operativo. Este retardo puede ser del orden de varios milisegundos (Figura 10.4c). Mientras este último enfoque puede ser adecuado para algunas aplicaciones de tiempo real, no será suficiente para aplicaciones más exigentes. En aquellos casos, la opción que se ha tomado se denomina a veces como expulsión inmediata. En este caso, el sistema operativo responde a una interrupción casi inmediatamente, a menos que el sistema esté en una sección bloqueada de código crítico. Los retardos de planificación para tareas de tiempo real pueden reducirse a 100 μs o menos.

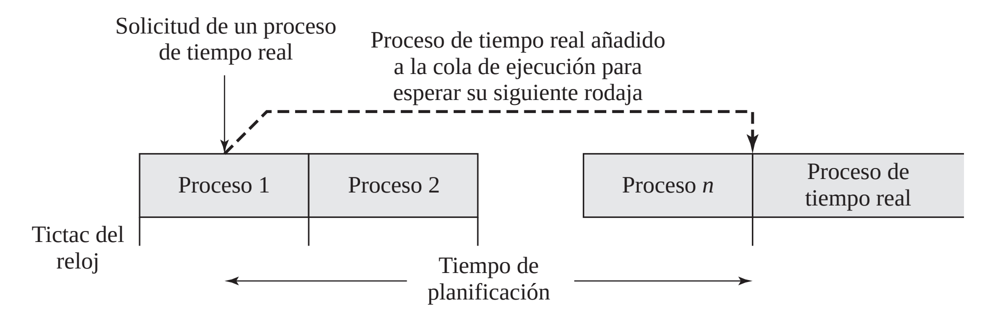
PLANIFICACIÓN DE TIEMPO REAL

La planificación de tiempo real es una de las áreas más activas de investigación en informática. En esta subsección, se ofrece una visión general de varios enfoques a la planificación de tiempo real y se muestran dos clases de algoritmos de planificación populares.

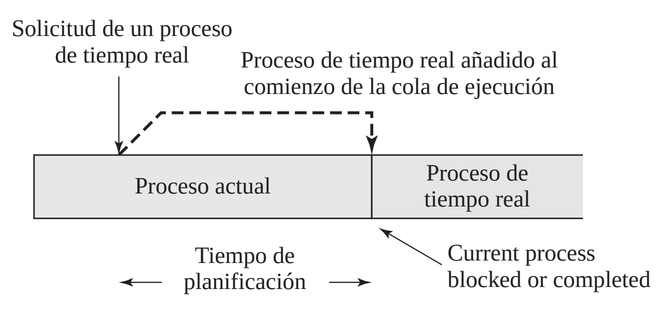
En un compendio de algoritmos de planificación de tiempo real, [RAMA94] se observa que los distintos enfoques de la planificación dependen de (1) cuando el sistema realiza análisis de planificabilidad; y si lo hace, de (2) si se realiza estática o dinámicamente; y de (3) si el resultado del análisis produce un plan de planificación de acuerdo al cual se desarrollarán las tareas en tiempo de ejecución. En base a estas consideraciones los autores identifican las siguientes clases de algoritmos:

- **Enfoques estáticos dirigidos por tabla.** En éstos se realiza un análisis estático de la factibilidad de la planificación. El resultado del análisis es una planificación que determina cuando, en tiempo de ejecución, debe comenzar a ejecutarse cada tarea.
- **Enfoques estáticos expulsivos dirigidos por prioridad.** También se realiza un análisis estático, pero no se obtiene una planificación. En cambio, el análisis se utiliza para asignar prioridades a las tareas, y así puede utilizarse un planificador expulsivo tradicional basado en prioridades.
- **Enfoques dinámicos basados en un plan.** La factibilidad se determina en tiempo de ejecución (dinámicamente) en vez de antes de comenzar la ejecución (estáticamente). Una nueva tarea será aceptada como ejecutable sólo si es posible satisfacer sus restricciones de tiempo. Uno de los resultados del análisis de factibilidad es un plan que se usará para decidir cuándo poner en marcha la tarea.
- **Enfoques dinámicos de mejor esfuerzo.** No se realiza análisis de factibilidad. El sistema intenta cumplir todos los plazos y aborta la ejecución de cualquier proceso cuyo plazo haya fallado.

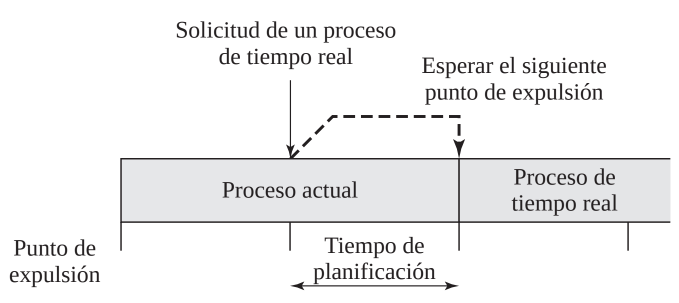
La **planificación estática dirigida por tabla** es aplicable a tareas que son periódicas. Los datos de entrada para el análisis son: tiempo periódico de llegada, tiempo de ejecución, plazo periódico de finalización, y prioridad relativa de cada tarea. El planificador intenta encontrar un plan que le permita cumplir todos los requisitos de todas las tareas periódicas. Éste es un enfoque predecible pero no



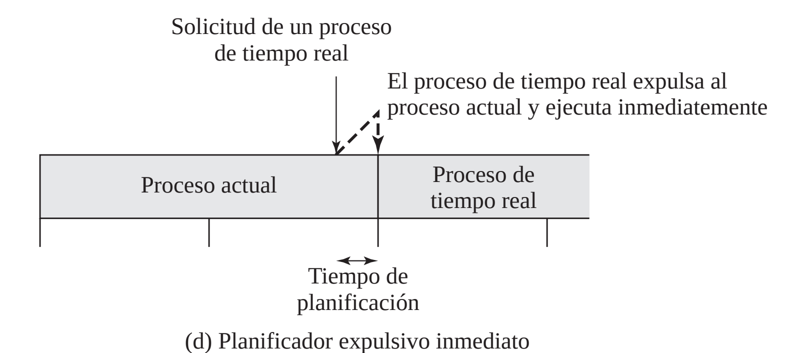
(a) Planificador expulsivo de turno circular



(b) Planificador no expulsivo dirigido por prioridad



(c) Planificador expulsivo dirigido por prioridad en puntos de expulsión



(d) Planificador expulsivo inmediato

Figura 10.4. Planificación de procesos de tiempo real.

flexible, dado que un cambio en cualquiera de los requisitos de las tareas requiere rehacer toda la planificación. El plazo más cercano primero, y otras técnicas de plazos periódicos (expuestas posteriormente) son típicas de esta categoría de algoritmos de planificación.

La **planificación estática con expulsión dirigida por prioridad** hace uso del mecanismo de planificación expulsivo dirigido por prioridades común a la mayoría de los sistemas multiprogramados que no son de tiempo real. En un sistema que no es de tiempo real, pueden utilizarse en múltiples factores para determinar la prioridad. Por ejemplo, en un sistema de tiempo compartido, la prioridad de un proceso puede cambiar dependiendo de qué consume más, CPU o E/S. En un sistema de tiempo real, la asignación de prioridades está relacionada con las restricciones de tiempo asociadas a cada tarea. Un ejemplo de este enfoque es el algoritmo de tasa monótona (expuesto posteriormente), que asigna prioridades estáticas a las tareas basándose en sus longitudes y sus periodos.

Con la **planificación dinámica basada en un plan**, cuando llega una nueva tarea, pero antes de que comience su ejecución, se intentará crear un plan que contenga las tareas previamente planificadas así como la nueva. Si la tarea recién llegada puede ser planificada de manera que se cumplan sus plazos sin que ninguna otra tarea planificada anteriormente pierda un plazo, la nueva tarea será aceptada poniéndose en marcha el nuevo plan de planificación.

La **planificación dinámica de mejor esfuerzo** es en enfoque utilizado en muchos sistemas operativos de tiempo real disponibles comercialmente hoy en día. Cuando llega una tarea, el sistema le asigna una prioridad basada en las características de la misma. De forma característica se utiliza algún tipo de planificación basada en plazos como la planificación del plazo más cercano. Así, las tareas no son periódicas y por tanto no es posible realizar un análisis estático de planificabilidad. Con este tipo de planificación, no sabremos si una determinada restricción de tiempo será satisfecha hasta que venza su plazo o la tarea se complete. Esta es la principal desventaja de esta forma de planificación. La ventaja es que es fácil de implementar.

PLANIFICACIÓN POR PLAZOS

La mayor parte de los sistemas operativos de tiempo real actuales están diseñados con el objetivo de arrancar las tareas de tiempo real tan rápidamente como sea posible, y por tanto enfatizan la manipulación rápida de las interrupciones y la activación rápida de las tareas. De hecho, esta no es una medida útil para la evaluación de los sistemas operativos de tiempo real. Generalmente, las aplicaciones de tiempo-real no se preocupan tanto de la velocidad de ejecución como de completar (o comenzar) sus tareas en los momentos más adecuados, ni muy pronto ni muy tarde, a pesar de la demanda dinámica de recursos u otros conflictos, sobrecarga de procesamiento y fallos hardware o software. Se deduce que las prioridades proporcionan una herramienta burda que no contempla el requisito de completar (o iniciar) la tarea en el momento más adecuado.

Ha habido varias propuestas de enfoques más potentes y apropiados para la planificación de tareas de tiempo real. Todos ellos se basan en tener información adicional acerca de cada tarea. En su forma más general, puede utilizarse la siguiente información de cada tarea:

- **Tiempo de activación.** Momento en el cual la tarea pasa a estar lista para ejecutar. En el caso de una tarea repetitiva o periódica se tratará de una secuencia de tiempos conocida de antemano. En el caso de una tarea aperiódica, este tiempo puede ser conocido previamente, o el sistema operativo sólo podrá ser consciente de la tarea cuando ésta pase a estar lista.
- **Plazo de comienzo.** Momento en el cual la tarea debe comenzar.
- **Plazo de conclusión.** Momento para el cual la tarea debe estar completada. Las aplicaciones de tiempo real típicas tendrán plazos de comienzo o bien plazos de conclusión, pero no ambos.

- **Tiempo de proceso.** Tiempo necesario para ejecutar la tarea hasta su conclusión. En algunos casos estará disponible, en otros, el sistema operativo lo estimará como una media exponencial (definida en el Capítulo 9), y aún en otros sistemas de planificación, esta información no se utiliza.
- **Recursos requeridos.** Conjunto de recursos (distintos del procesador) que la tarea necesita durante su ejecución.
- **Prioridad.** Mide la importancia relativa de la tarea. Las tareas de tiempo real duro pueden tener una prioridad «absoluta», provocando que el sistema falle si algún plazo no se cumple. Si el sistema debe continuar ejecutando sin importar lo que suceda, entonces pueden asignarse prioridades relativas, tanto a las tareas de tiempo real duro como suave, que el planificador utilizará como guía.
- **Estructura de subtareas.** Una tarea puede ser descompuesta en una subtarea obligatoria y una subtarea opcional. Sólo la subtarea obligatoria posee un plazo duro.

Hay varias dimensiones en la función de planificación de tiempo real en las que se tienen en consideración los plazos: qué tarea ejecutar a continuación, y qué tipo de expulsión se permite. Puede demostrarse que, para una estrategia de expulsión dada y utilizando o bien plazos de comienzo o bien de conclusión, la política de planificar la tarea de plazo más cercano minimiza la cantidad de tareas que fallan en sus plazos [BUTT99, HONG89, PANW88]. Este resultado se cumple tanto en configuraciones de procesador único como multiprocesador.

La expulsión es el otro factor crítico de diseño. Cuando se especifican los plazos de comienzo, entonces tiene sentido una planificación no expulsiva. En este caso, será responsabilidad de la tarea de tiempo real bloquearse a sí misma después de completar la porción crítica u obligatoria de su ejecución, para permitir que se satisfagan otros plazos de comienzo de tiempo real. Esto encaja con el patrón de la Figura 10.4b. Para un sistema con plazos de conclusión, es más apropiada una estrategia expulsiva (Figura 10.4c o d). Por ejemplo, si la tarea X está ejecutando y la tarea Y esta lista, puede haber circunstancias en las cuales la única manera de conseguir que tanto X como Y cumplan sus plazos de conclusión sea expulsar a X, ejecutar Y hasta finalizar, y luego retomar X hasta finalizar.

Tabla 10.2. Perfil de ejecución de dos tareas periódicas.

Proceso	Tiempo de llegada	Tiempo de ejecución	Plazo de conclusión
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
• • •	• • •	• • •	• • •
B(1)	0	25	50
B(2)	50	25	100
• • •	• • •	• • •	• • •

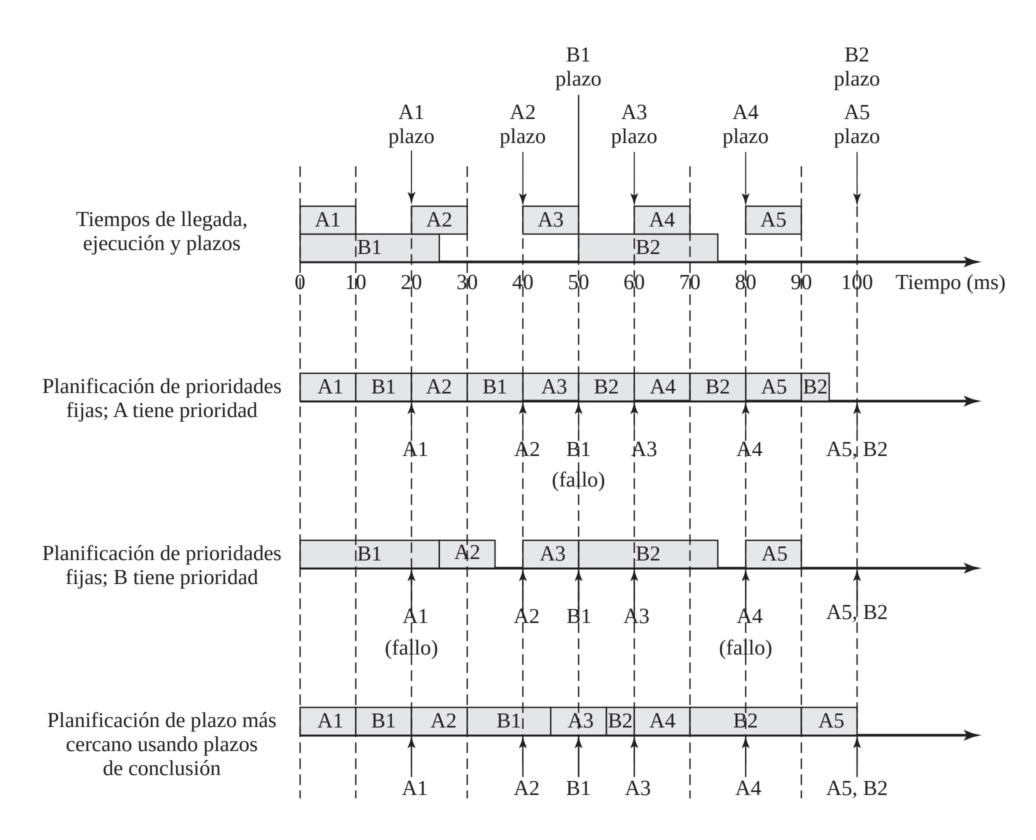


Figura 10.5. Planificación de tareas periódicas de tiempo real con plazos de conclusión (basado en la Tabla 10.2).

Como ejemplo de la planificación de tareas periódicas con plazos de conclusión, considere un sistema que recolecta y procesa datos de dos sensores, A y B. El plazo para la recolección de datos del sensor A es cada 20 ms, y para el sensor B cada 50 ms. Lleva 10 ms, incluyendo la sobrecarga del sistema operativo, procesar cada muestra de datos de A y 25 ms procesar cada muestra de datos de B. La Tabla 10.2 resume el perfil de ejecución de las dos tareas. La Figura 10.5 compara tres técnicas de planificación utilizando el perfil de ejecución de la Tabla 10.2. La primera fila de la Figura 10.5 repite la información de la Tabla 10.2; las restantes tres filas ilustran tres técnicas de planificación.

El computador es capaz de realizar una decisión de planificación cada 10 ms. Suponga que, bajo estas circunstancias, intentamos usar un esquema de planificación por prioridad. Los primeros dos diagramas de tiempo de la Figura 10.5 muestran el resultado. Si A tiene mayor prioridad, a la primera instancia de la tarea B sólo se le dan 20 ms de tiempo de proceso, en dos trozos de 10 ms, hasta el momento en que su plazo vence y por tanto falla. Si se le da mayor prioridad a B, entonces A fallará en su primer plazo. El último diagrama de tiempo muestra el uso de la planificación de plazo más cercano. En el instante $t = 0$, llegan A1 y B1. Dado que A1 tiene el plazo más cercano, es planificada primero. Cuando A1 se completa, el procesador se le entrega a B1. En $t = 20$, llega A2. Dado que A2 tiene un plazo más cercano que B1, B1 es interrumpida y A1 puede ejecutar hasta concluir. Entonces, en $t = 30$, se retoma B1. En t = 40, llega A3. Sin embargo, B1 tiene el plazo más cercano y se le permite continuar ejecutando hasta concluir en $t = 45$. A3 toma entonces el procesador y finaliza en $t = 55$.

En este ejemplo, planificando en cada punto de expulsión dando prioridad a la tarea con el plazo más próximo, pueden satisfacerse todos los requisitos del sistema. Dado que todas las tareas son periódicas y predecibles se utiliza un enfoque de planificación estática dirigida por una tabla.

Considere ahora un esquema para tratar con tareas aperiódicas con plazos de comienzo. La parte superior de la Figura 10.6 muestra los tiempos de llegada y plazos de comienzo para un ejemplo consistente en cinco tareas cada una de las cuales tiene un tiempo de ejecución de 20 ms. La Tabla 10.3 resume el perfil de ejecución de las cinco tareas.

Tabla 10.3. Perfil de ejecución de cinco tareas aperiódicas.

Proceso	Tiempo de llegada	Tiempo de ejecución	Plazo de comienzo
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

Un esquema directo es planificar siempre la tarea lista con el plazo más cercano y permitir que la tarea ejecute hasta concluir. Cuando se utiliza este enfoque en el ejemplo de la Figura 10.6, observe que aunque la tarea B precisa servicio inmediato, se le deniega el servicio. Este es el riesgo de

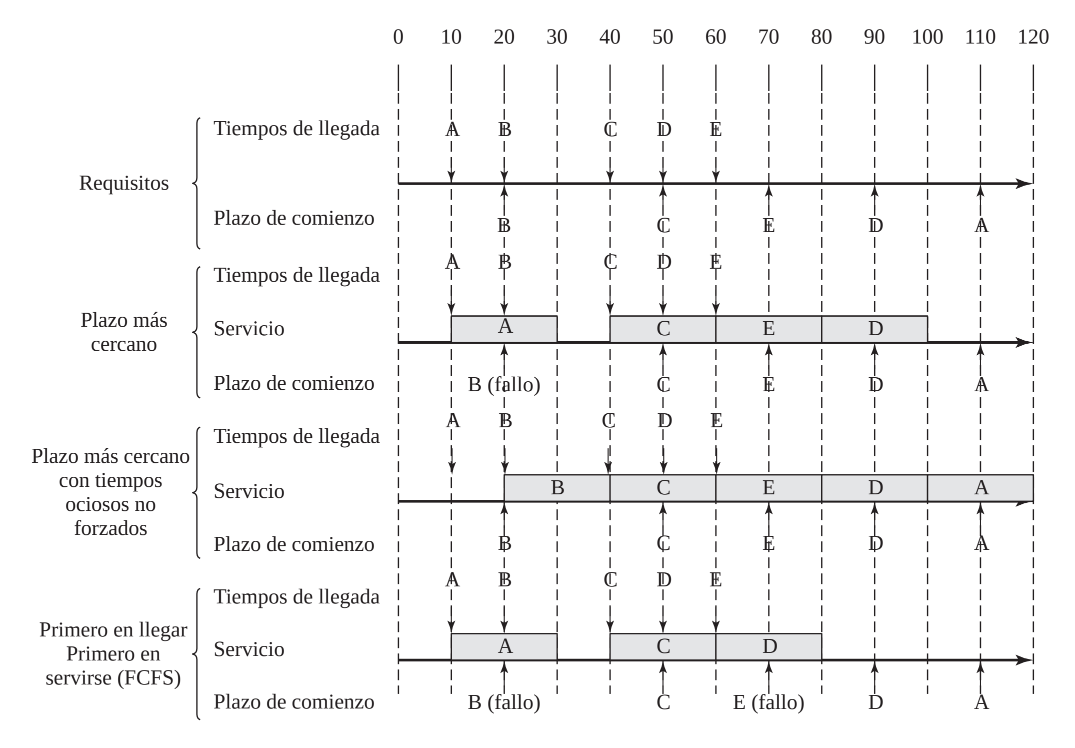


Figura 10.6. Planificación de tareas aperiódicas de tiempo real con plazos de comienzo.

manejar tareas aperiódicas, especialmente con plazos de comienzo. Una mejora de esta política mejorará el rendimiento si pueden conocerse con anticipación los plazos de una tarea antes de que ésta esté lista. Esta política, conocida como plazo más cercano con tiempos ociosos no forzados, funciona como sigue: siempre planificar la tarea elegible con el plazo más cercano y permitir que la tarea ejecute hasta concluir. Una tarea elegible puede no estar lista, y esto puede tener como resultado que el procesador permanezca ocioso aunque haya tareas listas. Observe que en el ejemplo el sistema evita la planificación de la tarea A aunque ésta sea la única tarea lista. El resultado es que, aunque el procesador no se utiliza con la máxima eficiencia, se satisfacen todos los requisitos de planificación. Finalmente, para comparar, se muestra la política FCFS. En este caso, las tareas B y E no cumplen sus plazos.

PLANIFICACIÓN DE TASA MONÓTONA

Uno de los métodos más prometedores para resolver los conflictos de planificación multitarea para tareas periódicas es la planificación de tasa monótona (RMS). El esquema fue propuesto por primera vez en [LIU73] pero sólo recientemente ha ganado popularidad [BRIA99, SHA94]. RMS asigna prioridades a las tareas en base a sus periodos.

La Figura 10.7 ilustra los parámetros relevantes de las tareas periódicas. El periodo de la tarea, T, es la cantidad de tiempo entre la llegada de una instancia de la tarea y la llegada de la siguiente instancia de la tarea. La tasa de una tarea (en Hercios) es simplemente el inverso de su periodo (en segundos). Por ejemplo, una tarea con un periodo de 50 ms ocurre a una tasa de 20 Hz. Típicamente, el final del periodo de una tarea es también el plazo crítico de la tarea, aunque algunas tareas pueden tener plazos más cercanos. El tiempo de ejecución (o de cómputo), C, es la cantidad de tiempo de proceso requerido por cada ocurrencia de la tarea. Debe quedar claro que en un sistema monoprocesador, el tiempo de ejecución no puede ser mayor que el periodo (es obligado $C \leq T$). Si una tarea periódica siempre ejecuta hasta que concluye, esto es, si a ninguna instancia de la tarea se le deniega nunca servicio por recursos insuficientes, entonces la utilización del procesador por esta tarea es $U = C/T$. Por ejemplo, si una tarea tiene un periodo de 80 ms y un tiempo de cómputo de 55 ms, su utilización de procesador será 55/80 = 0,6875.

Para el RMS la tarea de mayor prioridad es aquélla con el periodo más breve, la segunda tarea de mayor prioridad es aquélla con el segundo periodo más breve, y así sucesivamente. Cuando hay más de una tarea disponible para su ejecución, aquella con el periodo más breve se sirve primero. Si dibujamos la prioridad de las tareas como una función de su tasa, el resultado es una función monótonamente creciente (Figura 10.8); de ahí el nombre, planificación de tasa monótona.

Una medida de la efectividad de un algoritmo de planificación periódica es si se garantiza o no que se cumplen todos los plazos duros. Suponga que tenemos n tareas, cada una con periodo y tiempo de cómputo fijos. Para que sea posible cumplir todos los plazos, debe cumplirse la siguiente relación:

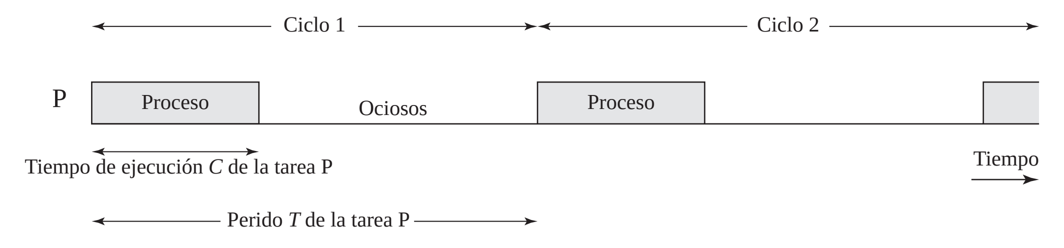


Figura 10.7. Diagrama de tiempos de una tarea periódica.

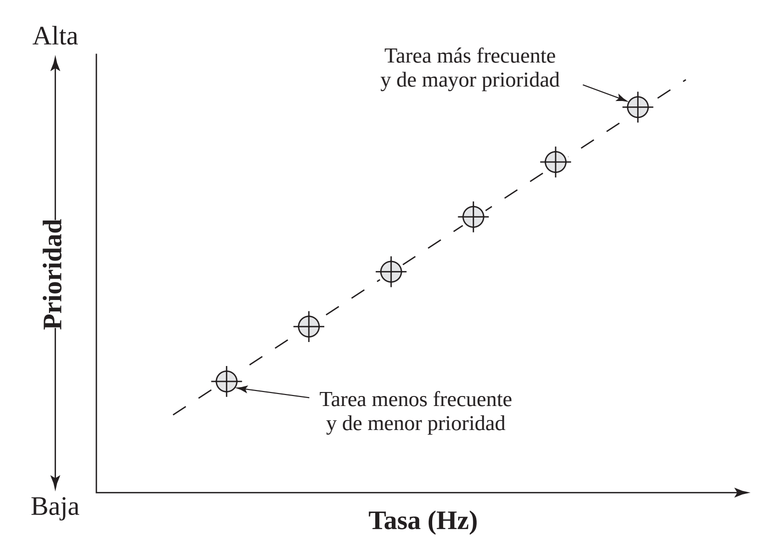


Figura 10.8. Un conjunto de tareas con RMS [WARR91].

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq 1 \qquad (10.1)$$

La suma de la utilización del procesador de las tareas no puede exceder un valor de 1, que se corresponde con la utilización total del procesador. La Ecuación (10.1) proporciona un límite al número de tareas que un algoritmo de planificación perfecto puede planificar satisfactoriamente. Para un algoritmo en particular, el límite puede ser menor. Para el RMS, puede mostrarse que se cumple la siguiente relación:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \qquad (10.2)$$

Tabla 10.4. Valor del límite superior del RMS.

n	$n(2^{1/n} - 1)$
1	1,0
2	0,828
3	0,779
4	0,756
5	0,743
6	0,734
•	•
•	•
•	•
∞	ln 2 ≈ 0,693

La Tabla 10.4 muestra algunos valores para este límite superior. A medida que el número de tareas aumenta, el límite de planificabilidad converge a ln 2 ≈ 0,693.

Como ejemplo, considere el caso de tres tareas periódicas, donde $U_i = C_i/T_i$:

- Tarea P_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0,2$
- Tarea P_2: $C_2 = 40$; $T_2 = 150$; $U_2 = 0,267$
- Tarea P_3: $C_3 = 100$; $T_3 = 350$; $U_3 = 0,286$

La utilización total de estas tres tareas es 0,2 + 0,267 + 0,286 = 0,753. El límite superior para la planificabilidad de estas tres tareas usando RMS es

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \leq 3(2^{1/3} - 1) = 0,779$$

Dado que la utilización total necesaria para estas tres tareas es menor que el límite superior para RMS (0,753 < 0,779), se sabe que si se utiliza RMS, todas las tareas serán planificadas satisfactoriamente.

También puede demostrarse que el límite superior de la Ecuación (10.1) se cumple para la planificación del plazo más cercano. Así, es posible conseguir una mayor utilización del procesador y por tanto acomodar más tareas periódicas con la planificación del plazo más cercano. No obstante, RMS ha sido ampliamente adoptado para su uso en aplicaciones industriales. [SHA91] ofrece la siguiente explicación:

1. En la práctica, la diferencia de prestaciones es pequeña. El límite superior de la Ecuación (10.2) es conservador y, en la práctica, a menudo se consiguen utilizaciones tan altas como el 90%.
2. La mayoría de los sistemas de tiempo real duro también tienen componentes de tiempo real suave, como ciertos mensajes no críticos o auto comprobaciones que pueden ser ejecutadas en niveles de prioridad menores para absorber el tiempo de procesador que no se utiliza con la planificación RMS de tareas de tiempo real duro.
3. Es más fácil conseguir la estabilidad con RMS. Cuando un sistema no puede cumplir todos los plazos por sobrecarga o errores transitorios, los plazos de las tareas esenciales pueden ser garantizados si este subconjunto de tareas es planificable. En un enfoque de asignación estática de prioridades, sólo se necesita asegurar que las tareas esenciales tienen prioridades relativamente altas. Con la planificación del plazo más cercano, la prioridad de una tarea periódica cambia de un periodo a otro. Esto hace más difícil asegurar que las tareas esenciales cumplen sus plazos.

INVERSIÓN DE PRIORIDAD

La inversión de prioridad es un fenómeno que puede suceder en cualquier esquema de planificación expulsivo basado en prioridades, pero es particularmente relevante en el contexto de la planificación de tiempo real. El caso más conocido de inversión de prioridad sucedió en la misión Mars Pathfinder. Este vehículo robot alcanzó Marte el 4 de julio de 1997, y comenzó a recolectar y transmitir voluminosas cantidades de datos a la Tierra. Pero a los pocos días de la misión, el software de abordo comenzó a experimentar reinicializaciones totales del sistema, con las consiguientes pérdidas de datos. Después de mucho esfuerzo por parte del equipo del Jet Propulsion Laboratory que construyó el Pathfinder, se descubrió que el problema tenía que ver con la inversión de prioridad [JONE97].

En cualquier esquema de planificación por prioridad, el sistema debe siempre estar ejecutando la tarea de mayor prioridad. La **inversión de prioridad** sucede cuando las circunstancias dentro de sistema fuerzan a una tarea de mayor prioridad a esperar por una tarea de menor prioridad. Un ejemplo sencillo de inversión de prioridad sucede si una tarea de menor prioridad ha bloqueado un recurso (un dispositivo o un semáforo binario) y una tarea de mayor prioridad intenta bloquear el mismo recurso. La tarea de mayor prioridad pasará a estado bloqueado hasta que el recurso esté disponible. Si la tarea de menor prioridad termina pronto con el recurso y lo libera, la tarea de mayor prioridad puede ser retomada rápidamente y es posible que no se violen restricciones de tiempo real.

Una situación más seria, es la conocida como **inversión de prioridad ilimitada**, en la cual la duración de la inversión de prioridad depende no sólo del tiempo necesario para conseguir el recurso compartido sino también de las acciones impredecibles de otras tareas no relacionadas. La inversión de prioridad que sucedió en el software del Pathfinder fue ilimitada y sirve como un buen ejemplo de este fenómeno. La exposición siguiente se basa en [TIME02]. El software del Pathfinder incluye las siguientes tres tareas, en orden de prioridad decreciente:

T_1. Comprueba periódicamente la salud del software y los sistemas de la nave espacial.

T_2. Procesa datos de imágenes.

T_3. Realiza ocasionalmente una comprobación del estado de los equipos.

Después de ejecutarse T_1, se reinicializa un temporizador a su valor máximo. Si este temporizador vence alguna vez, se asume que la integridad del software del vehículo se ha visto comprometida de algún modo. El procesador se para, se reinicializan todos los dispositivos, se recarga completamente el software, se verifican los sistemas de la nave espacial, y el sistema arranca de nuevo. Ésta secuencia de recuperación no se completa hasta el siguiente día. T_1 y T_3 comparten una estructura de datos, protegida por un semáforo binario `s`. La Figura 10.9a muestra la secuencia que causa la inversión de prioridad:

t_1. T_3 comienza a ejecutar.

t_2. T_3 bloquea el semáforo `s` y entra en su sección crítica.

t_3. T_1, que tiene mayor prioridad que T_3, expulsa a T_3 y comienza a ejecutar.

t_4. T_1 intenta entrar en su sección crítica pero se bloquea porque el semáforo está en propiedad de T_3, T_3 retoma su ejecución en su sección crítica.

t_5. T_2, que tiene mayor prioridad que T_3, expulsa a T_3 y comienza a ejecutar.

t_6. T_2 se suspende por alguna causa no relacionada con T_1 ni T_2, y se retoma T_3.

t_7. T_3 abandona su sección crítica y desbloquea el semáforo. T_1 expulsa a T_3, bloquea el semáforo y entra en su sección crítica.

En esta secuencia de circunstancias, T_1 debe esperar a que T_3 y T_2 concluyan, y no alcanza a reinicializar el temporizador antes de que expire.

En sistemas prácticos, se utilizan dos enfoques alternativos para evitar la inversión de prioridad ilimitada: el protocolo de herencia de prioridad y el protocolo de techo de prioridad.

La idea básica de la **herencia de prioridad** es que una tarea de menor prioridad heredará la prioridad de cualquier tarea de mayor prioridad pendiente de un recurso que compartan. Este cambio de prioridad sucede tan pronto la tarea de mayor prioridad se bloquea en el recurso; y debe finalizar cuando la tarea de menor prioridad libera el recurso. La Figura 10.9b muestra como la herencia de prioridad resuelve el problema de la inversión de prioridad ilimitada ilustrado en la Figura 10.9a. La secuencia de los eventos relevantes es como sigue:

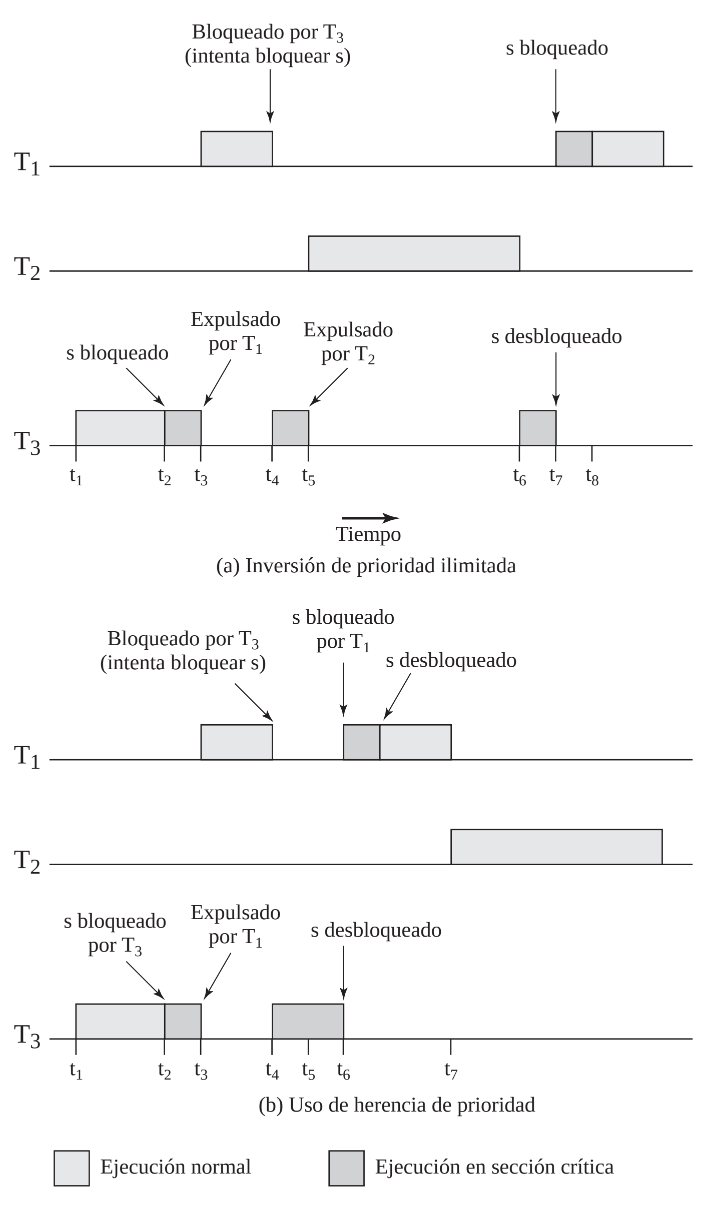


Figura 10.9. Inversión de prioridad.

t_1. T_3 comienza a ejecutar.

t_2. T_3 bloquea el semáforo s y entra en su sección crítica.

t_3. T_1, que tiene mayor prioridad que T_3, expulsa a T_3 y comienza a ejecutar.

t_4. T_1 intenta entrar en su sección crítica pero se bloquea porque el semáforo está en propiedad de T_3. A T_3 se le asigna inmediatamente, pero de manera temporal, la misma prioridad de T_1. T_3 retoma su ejecución en su sección crítica.

t_5. T_2, está listo para ejecutar pero, dado que ahora T_3 tiene mayor prioridad, T_2 no puede expulsar a T_3.

t_6. T_3 abandona su sección crítica y desbloquea el semáforo: su prioridad se reduce a la que tenía anteriormente. T_1 expulsa a T_3, bloquea el semáforo y entra en su sección crítica.

t_7. T_1 se suspende por alguna causa no relacionada con T_2, y T_2 comienza a ejecutar.

En el enfoque de **techo de prioridad**, se asocia una prioridad con cada recurso. La prioridad asociada con un recurso es un nivel más alta que la prioridad de su usuario más prioritario. Entonces, el planificador asigna esta prioridad a cualquier tarea que acceda al recurso. Cuando la tarea termina de usar el recurso, su prioridad se restablece al valor normal.

10.3. PLANIFICACIÓN EN LINUX

En Linux 2.4 y anteriores, Linux tiene capacidad para la planificación de tiempo real así como un planificador para procesos no de tiempo real que hace uso del algoritmo de planificación tradicional de UNIX descrito en la Sección 9.3. Linux 2.6 incluye esencialmente la misma capacidad de planificación de tiempo real que las ediciones previas y un planificador sustancialmente modificado para procesos no de tiempo real. Examinemos estas dos áreas por turno.

PLANIFICACIÓN DE TIEMPO REAL

Las tres clases de planificación en Linux son las siguientes:

- **SCHED_FIFO.** Hilos de tiempo real planificados FIFO.
- **SCHED_RR.** Hilos de tiempo real planificados con turno circular.
- **SCHED_OTHER.** Otros hilos no de tiempo real.

Dentro de cada clase, pueden utilizarse múltiples prioridades, siendo las prioridades de las clases de tiempo real mayores que las prioridades para la clase SCHED_OTHER. Los valores por omisión son los siguientes: la prioridad de las clases de tiempo real van del 0 al 99 inclusive, y para la clase SCHED_OTHER van del 100 al 139. Un número menor significa mayor prioridad.

Para los hilos FIFO, se aplican las siguientes reglas:

1. El sistema no interrumpirá un hilo FIFO en ejecución excepto en los siguientes casos:
   a) Otro hilo FIFO de mayor prioridad pasa a estado listo.
   b) El hilo FIFO en ejecución pasa a estar bloqueado en espera por algún evento, como E/S.
   c) El hilo FIFO en ejecución cede voluntariamente el procesador realizando una llamada a la primitiva sched_yield.
2. Cuando un hilo FIFO en ejecución es interrumpido, se le sitúa en una cola asociada con su prioridad.
3. Cuando un hilo FIFO pasa a estar listo y ese hilo es de mayor prioridad que el hilo actualmente en ejecución, entonces el hilo actualmente en ejecución es expulsado y el hilo FIFO listo de mayor prioridad es ejecutado. Si hay más de un hilo que tiene esa mayor prioridad, se escogerá el hilo que lleve más tiempo esperando.

La política SCHED_RR es similar a la política SCHED_FIFO, excepto por el añadido de una rodaja de tiempo asociada con cada hilo. Cuando un hilo SCHED_RR ha estado ejecutando durante toda su rodaja de tiempo, se le suspende y se selecciona para su ejecución un hilo de tiempo real de igual o mayor prioridad.

La Figura 10.10 es un ejemplo que ilustra la diferencia entre las planificaciones FIFO y RR. Asúmase un proceso con cuatro hilos con tres prioridades relativas asignadas como se muestra en la Figura 10.10a. Asúmase que todos los hilos en espera están listos para ejecutar cuando el hilo actual pasa a esperar o termina y que ningún hilo de mayor prioridad es despertado mientras el hilo está ejecutando. La Figura 10.10b muestra una secuencia en la que todos los hilos pertenecen a la clase SCHED_FIFO. El hilo D ejecuta hasta que espera o termina. Luego, aunque los hilos B y C tienen la misma prioridad, arranca el hilo B porque lleva esperando más tiempo que el hilo C. El hilo B ejecuta hasta que espera o termina; luego el hilo C ejecuta hasta que espera o termina. Finalmente, ejecuta el hilo A.

La Figura 10.10c muestra un ejemplo de secuencia si todos los hilos perteneciesen a la clase SCHED_RR. El hilo D ejecuta hasta que espera o termina. Luego, los hilos B y C se entrelazan en el tiempo, porque ambos tienen la misma prioridad. Finalmente, ejecuta el hilo A.

La última clase de planificación es SCHED_OTHER. Un hilo en esta clase sólo puede ejecutar si no hay hilos de tiempo real listos para ejecutar.

PLANIFICACIÓN NO DE TIEMPO REAL

El planificador de Linux 2.4 para la clase SCHED_OTHER no escalaba bien con un número creciente de procesadores y un número creciente de procesos. Para atacar este problema, Linux 2.6 usa un planificador completamente nuevo conocido como el planificador O(1)[4]. El planificador ha sido diseñado para que el tiempo de seleccionar el proceso adecuado y asignarlo a un procesador sea constante, sin importar la carga del sistema y el número de procesadores.

El núcleo mantiene dos estructuras de datos para la planificación por cada procesador del sistema, con la siguiente forma (Figura 10.11):

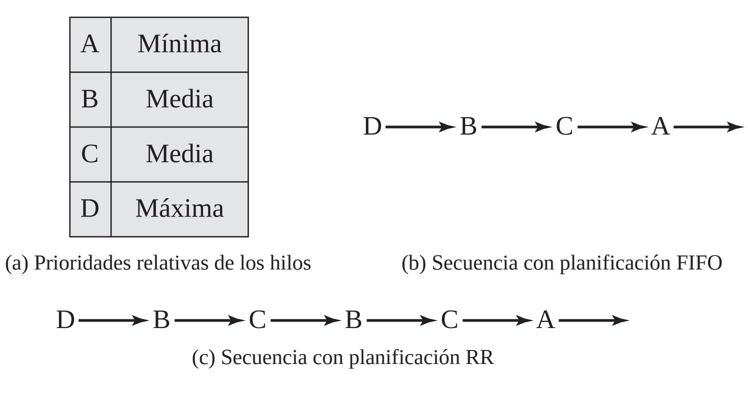


Figura 10.10. Ejemplo de la planificación de tiempo real de Linux.

---

[4] El término O(1) es un ejemplo de la notación «gran-O», usada para caracterizar la complejidad de los algoritmos. En el sitio web de este libro puede encontrarse un documento que explica esta notación.

```
struct prio_array {
    int                 nr_active;              /* número de tareas en este array */
    unsigned long       bitmap[BITMAP_SIZE];    /* bitmap de prioridades */
    struct list_head    queue[MAX_PRIO];        /* colas de prioridad */
```

Se mantiene una cola separada por cada nivel de prioridad. El número total de colas en la estructura es `MAX_PRIO` cuyo valor normalmente es 140. La estructura también incluye un array de bits de tamaño suficiente para proporcionar un bit por nivel de prioridad. Así, con 140 niveles de prioridad y palabras de 32-bit, `BITMAP_SIZE` tiene un valor de 5. Esto crea un vector de 160 bits, de los cuales 20 se ignoran. Este vector indica qué colas no están vacías. Finalmente, `nr_active` indica el número total de tareas presente en todas las colas. Se mantienen dos estructuras: una estructura de colas activas y una estructura de colas vencidas.

Inicialmente, ambos vectores se ponen a cero y todas las colas están vacías. Cuando un proceso pasa a estar listo, se le asigna a la cola de la prioridad apropiada en la estructura de colas activas y se le asigna la rodaja de tiempo apropiada. Si la tarea es expulsada antes de completar su rodaja de tiempo, se le devuelve a una cola activa. Cuando una tarea completa su rodaja de tiempo, va a la cola apropiada de la estructura de colas vencidas y se le asigna una nueva rodaja de tiempo. Toda la planificación se realiza entre las tareas en la estructura de colas activas. Cuando la estructura de colas activas está vacía la simple asignación de un puntero resulta en el cambio de colas activas a vencidas, y continúa la planificación.

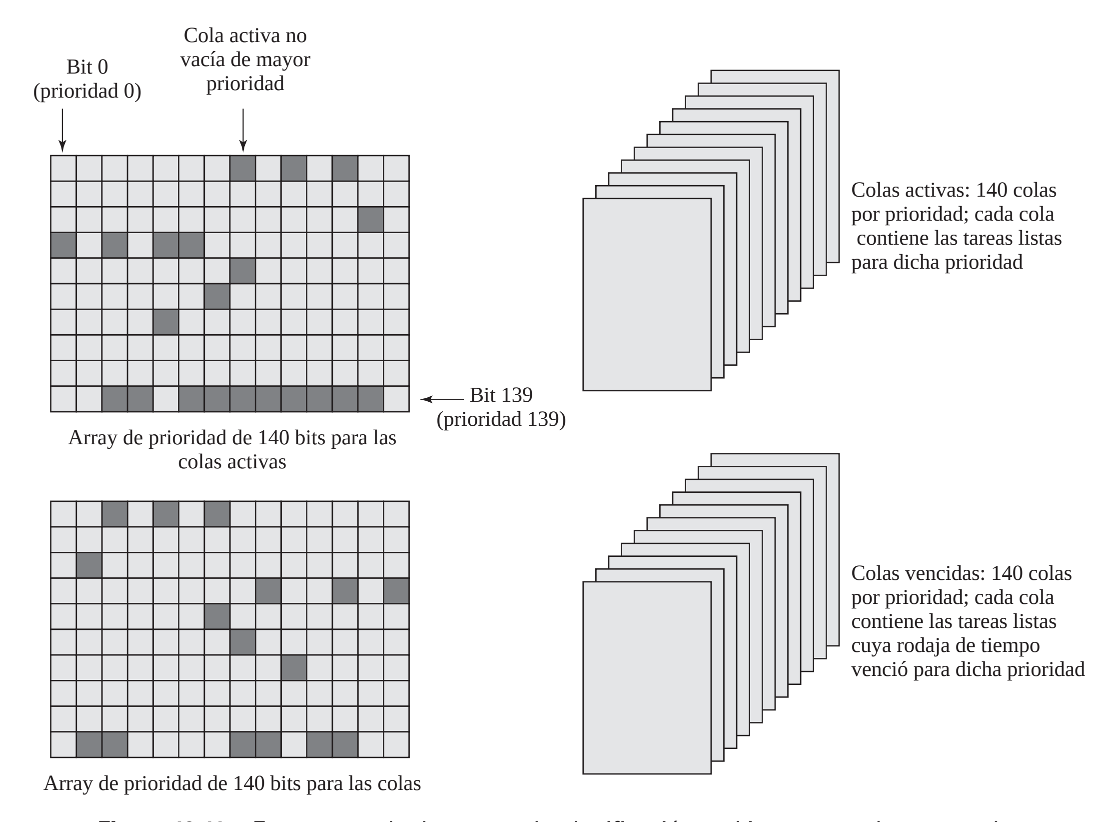


Figura 10.11. Estructuras de datos para la planificación en Linux por cada procesador.

La planificación es simple y eficiente. En un procesador dado, el planificador toma la cola no vacía de mayor prioridad. Si hay múltiples tareas en esa cola, las tareas se planifican de manera turno circular.

Linux también incluye un mecanismo para mover tareas de la lista de colas de un procesador a la de otro. Periódicamente, el planificador comprueba si hay un desbalanceo sustancial entre el número de tareas asignadas a cada procesador. Para balancear la carga, el planificador puede transferir algunas tareas. Las tareas activas de mayor prioridad se seleccionan para ser transferidas, dado que es más importante distribuir uniformemente las tareas de mayor prioridad.

Cálculo de prioridades y rodajas de tiempo A cada tarea no de tiempo real se le asigna una prioridad inicial en el rango de 100 a 139, por un valor por comisión de 120. Esta es la prioridad estática de la tarea que es especificada por el usuario. A medida que la tarea ejecuta, se calcula una prioridad dinámica como función de la prioridad estática de la tarea y del comportamiento de su ejecución. El planificador Linux está diseñado para favorecer las tareas limitadas por la E/S sobre las limitadas por el procesador. Esta preferencia tiende a proporcionar una buena respuesta interactiva. La técnica utilizada en el Linux para determinar la prioridad dinámica es guardar información sobre cuánto tiempo duermen los procesos (esperando por un evento) versus cuánto tiempo ejecutan. En esencia, a una tarea que pierde la mayor parte del tiempo durmiendo se le da la mayor prioridad.

Las rodajas de tiempo se asignan en el rango de los 10 ms a los 200 ms. En general a las tareas de mayor prioridad se le asignan rodajas de tiempo más largas.

Relación con las tareas de tiempo real Las tareas de tiempo-real se manipulan de manera distinta que las tareas no de tiempo real en las colas de prioridad. Se aplican las siguientes consideraciones:

1. Todas las tareas de tiempo real tienen solamente prioridad estática; no se realizaron cálculos de prioridad dinámica.
2. Las tareas SCHED_FIFO no tienen asignadas rodajas de tiempo. Estas tareas se planifican siguiendo la disciplina FIFO. Si una de estas tareas se bloquea, cuando se desbloquee volverá a la misma cola de prioridad en la lista de colas activas.
3. Aunque las tareas SCHED_RR tienen asignadas rodajas de tiempo, tampoco se les traslada nunca a la lista de colas vencidas. Cuando una de estas tareas consume toda su rodaja de tiempo, se le devuelve a su cola de prioridad con el mismo valor de rodaja de tiempo. Los valores de rodaja de tiempo nunca se cambian.

El efecto de estas reglas es que el cambio entre la lista de colas activas y la lista de colas vencidas sólo sucede cuando no hay tareas de tiempo real listas esperando para ejecutar.

10.4. PLANIFICACIÓN EN UNIX SVR4

El algoritmo de planificación utilizado en UNIX SVR4 es una revisión completa del algoritmo de planificación utilizado en los sistemas UNIX anteriores (descrito en la Sección 9.3). El nuevo algoritmo está diseñado para dar la mayor preferencia a los procesos de tiempo real, la siguiente mayor preferencia a los procesos en modo núcleo, y la menor preferencia a otros procesos en modo-usuario, conocidos como procesos de tiempo compartido.

Las dos principales modificaciones implementadas en SVR4 son las siguientes:

1. El añadido de un planificador de prioridad estática expulsivo y la introducción de un conjunto de 160 niveles de prioridad divididos en tres clases de prioridad.

2. La inserción de puntos de expulsión. Dado que el núcleo básico no es expulsivo, sólo puede ser dividido en pasos de proceso que deben ejecutar hasta su conclusión sin interrupción. Entre estos pasos de proceso, se han identificado ciertos puntos de expulsión donde el núcleo puede ser interrumpido de manera segura para planificar un nuevo proceso. Un punto seguro se define como una región de código donde todas las estructuras de datos del núcleo están actualizadas y son consistentes o bien están bloqueadas por un semáforo.

La Figura 10.12 ilustra los 160 niveles de prioridad definidos en SVR4. Cada proceso se define como perteneciente a una de estas tres clases de prioridad y se le asigna un nivel de prioridad dentro de la clase. Las clases son los siguientes:

- **Tiempo-real (159-100).** Los procesos en estos niveles de prioridad se garantiza que serán elegidos para ejecutar antes que cualquier proceso de núcleo o de tiempo compartido. Además, los procesos de tiempo real pueden hacer uso de los puntos de expulsión para expulsar procesos de núcleo y procesos de usuario.
- **Núcleo (99-60).** Los procesos en estos niveles de prioridad se garantiza que serán elegidos para ejecutar antes que cualquier proceso de tiempo compartido pero serán retrasados por los procesos de tiempo real.
- **Tiempo-compartido (59-0).** Son los procesos de más baja prioridad, pensados para las aplicaciones de usuario que no sean de tiempo real.

Clase de prioridad	Valor global	Secuencia de planificación
Tiempo real	159 • • • • 100	Primero ↓
Núcleo	99 • • 60	↓
Tiempo compartido	59 • • • • 0	↓ Último

Figura 10.12. Clases de prioridad de SVR4.

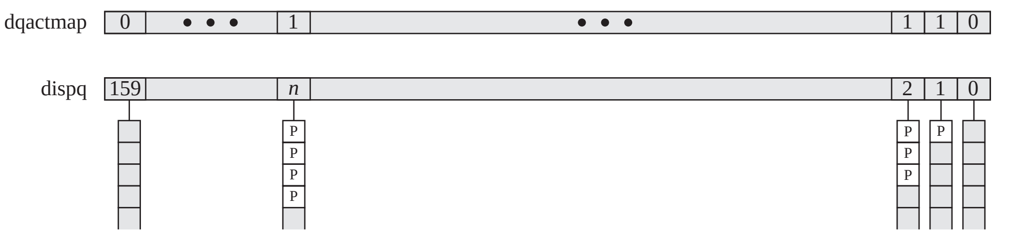


Figura 10.13. Colas de activación de SVR4.

La Figura 10.13 indica cómo está implementada la planificación en SVR4. Asociada con cada nivel de prioridad hay una cola de activación, y los procesos de un nivel de prioridad dado se ejecutan de manera turno circular. Un vector de bits, `dqactmap,` contiene un bit por cada nivel de prioridad; el bit se pone a uno por cada nivel de prioridad con cola no vacía. Cuando un proceso activo abandona el estado Ejecución, debido a un bloqueo, al vencimiento de su rodaja de tiempo, o por expulsión, el activador comprueba `dqactmap` y activa un proceso listo de la cola no vacía de mayor prioridad. Además, cuando se alcanza uno de los puntos de expulsión definidos, el núcleo comprueba una variable llamada `kprunrun.` Si no es 0, significa que al menos un proceso de tiempo real está en estado Listo, y el núcleo expulsa al proceso actual si es de menor prioridad que el proceso listo de tiempo real de mayor prioridad.

Dentro de la clase de tiempo compartido, la prioridad de los procesos es variable. El planificador reduce la prioridad de un proceso cada vez que utiliza totalmente su cuanto de tiempo, y eleva su prioridad si se bloquea en un evento o en un recurso. El cuanto de tiempo ubicado para un proceso de tiempo compartido depende de su prioridad, yendo de los 100 ms para la prioridad 0 a los 10 ms para la prioridad 59. Cada proceso de tiempo real tiene una prioridad fija y un cuanto de tiempo fijo.

10.5. PLANIFICACIÓN EN WINDOWS

Windows está diseñado para ser tan sensible como sea posible a las necesidades de un único usuario en un entorno altamente interactivo o en el papel de un servidor. Windows implementa un planificador expulsivo con un sistema flexible de niveles de prioridad que incluye planificación turno circular dentro de cada nivel y, para algunos niveles, variación dinámica de la prioridad sobre la base de la actividad de su hilo actual.

PRIORIDAD DE PROCESOS E HILOS

Las prioridades en Windows se organizan en dos bandas o clases: tiempo real y variable. Cada una de estas bandas consiste en 16 niveles de prioridad. Los hilos que precisan atención inmediata están en la clase de tiempo real, que incluye funciones como las comunicaciones y las tareas de tiempo real.

En general, dado que Windows hace uso de un planificador expulsivo basado en prioridades, los hilos con prioridades de tiempo real tienen preferencia sobre otros hilos. En un monoprocesador, cuando un hilo cuya prioridad es mayor que la del hilo actualmente en ejecución pasa a estar listo, el hilo menos prioritario es expulsado y se le entrega el procesador al hilo de mayor prioridad.

Las prioridades se manejan de manera un poco diferente en las dos clases (Figura 10.14). En la clase de prioridad de tiempo real, todos los hilos tienen una prioridad fija que nunca cambia. En la clase de prioridad variable, la prioridad del hilo comienza con algún valor asignado inicialmente y luego puede cambiar, arriba o abajo, durante la vida del hilo. Así, hay una cola FIFO en cada nivel de prioridad, pero un proceso puede migrar a una de las otras colas dentro de la clase de prioridad variable. No obstante, un hilo en el nivel de prioridad 15 no puede pasar al nivel 16 ni a ningún otro nivel de la clase de tiempo real.

La prioridad inicial de un hilo de la clase de prioridad variable se determina en base a dos cantidades: la prioridad base del proceso y la prioridad base del hilo. Uno de los atributos del objeto proceso es la prioridad base del proceso, que puede tomar cualquier valor de 0 a 15. Cada objeto hilo asociado con un objeto proceso tiene un atributo prioridad base del hilo que indica la prioridad base del hilo relativa a la del proceso. La prioridad base del hilo puede ser igual a la de su proceso o dentro de dos niveles por encima o por debajo de la del proceso. Así, por ejemplo, si un proceso tiene

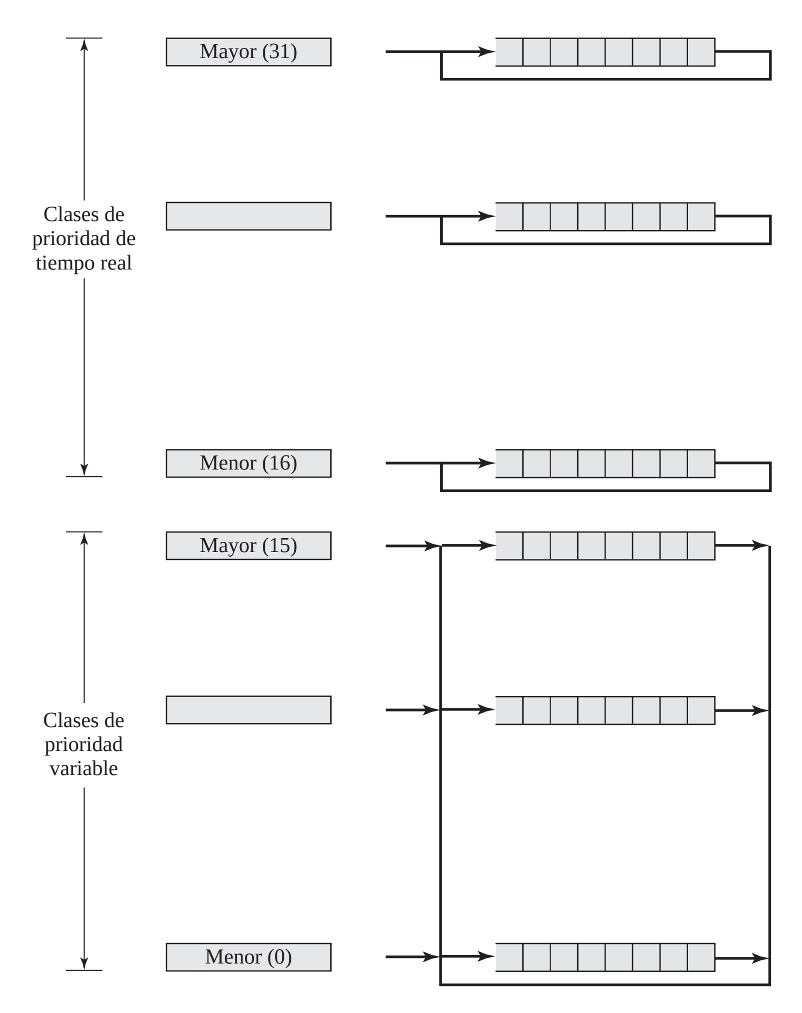


Figura 10.14. Prioridades de activación de hilos en Windows.

prioridad base de 4 y uno de sus hilos tiene una prioridad base de -1, entonces la prioridad inicial del hilo es 3.

Una vez que un hilo de la clase de prioridad variable ha sido activado, su prioridad real, conocida como prioridad dinámica del hilo, puede fluctuar dentro de unos límites dados. La prioridad dinámica nunca puede caer por debajo del rango inferior de la prioridad base del hilo y nunca podrá exceder 15. La Figura 10.15 ofrece un ejemplo. El objeto proceso tiene un atributo de prioridad base de 4. Cada objeto hilo asociado con este objeto proceso debe tener una prioridad inicial entre 2 y 6. La prioridad dinámica de cada hilo puede fluctuar en el rango de 2 a 15. Si un hilo es interrumpido porque ha usado completamente su cuanto de tiempo actual, el ejecutivo de Windows baja su prioridad. Si un hilo se interrumpe para esperar por un evento de E/S, el ejecutivo de Windows sube su prioridad. Así, los hilos limitados por procesador tienden a prioridades menores y los hilos limitados por E/S tienden a prioridades mayores. En el caso de los hilos limitados por E/S, el ejecutivo sube la prioridad más para esperas interactivas (por ejemplo, espera por teclado o pantalla) que por otro tipo de E/S (por ejemplo, E/S sobre disco). Así, los hilos interactivos tienden a tener las mayores prioridades dentro de la clase de prioridad variable.

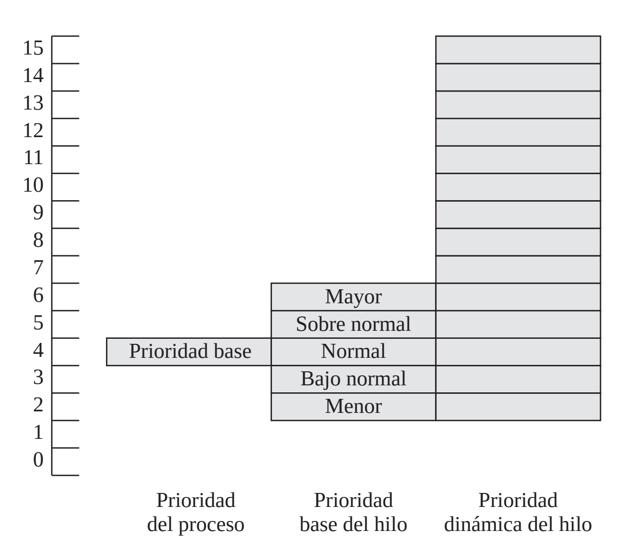


Figura 10.15. Ejemplo de relación entre las prioridades de Windows.

PLANIFICACIÓN MULTIPROCESADOR

Cuando Windows ejecuta en un procesador único, el hilo de mayor prioridad está siempre activo a menos que este esperando por un evento. Si hay más de un hilo que tiene la mayor prioridad, entonces se comparte el procesador, en turno circular, entre todos los hilos de ese nivel de prioridad. En un sistema multiprocesador con N procesadores, los (N - 1) hilos de mayor prioridad están siempre activos, ejecutando en exclusiva sobre los (N - 1) procesadores extra. El resto de los hilos, de menor prioridad, comparten el único procesador restante. Por ejemplo, si hay tres procesadores, los dos hilos de mayor prioridad ejecutan en dos procesadores, mientras que los restantes hilos ejecutan en el procesador restante.

La disciplina anterior se ve afectada por el atributo del hilo denominado afinidad de procesador. Si un hilo está listo para ejecutar pero los únicos procesadores disponibles no están en su conjunto de procesadores afines, el hilo será forzado a esperar y el ejecutivo planificará el siguiente hilo disponible.

10.6. RESUMEN

Con un multiprocesador fuertemente acoplado, múltiples procesadores tienen acceso a la misma memoria principal. En esta configuración, la estructura de planificación es de algún modo más compleja. Por ejemplo, un proceso dado puede ser asignado al mismo procesador para toda su vida o puede ser activado sobre cualquier procesador cada vez que entra en estado Ejecutable. Ciertos estudios de prestaciones sugieren que las diferencias entre varios algoritmos de planificación son menos significantes en sistemas multiprocesador.

Un proceso o tarea de tiempo real es aquel que ejecuta en conexión con algunos procesos o funciones o conjunto de eventos externos al sistema computador y que debe cumplir uno o más plazos para interactuar efectiva y correctamente con el entorno externo. Un sistema operativo de tiempo real es aquel capaz de manejar procesos de tiempo real. En este contexto, los criterios tradicionales de los algoritmos de planificación no tienen cabida. En cambio, el factor clave es cumplir los plazos de

tiempo. Los algoritmos apropiados en este contexto se basan fundamentalmente en la expulsión y en reaccionar a plazos de tiempo relativos.

10.7. LECTURAS RECOMENDADAS

[WEND89] expone de manera interesante los enfoques para la planificación multiprocesador. Un buen tratamiento de la planificación de tiempo real está contenido en [LIU00]. Las siguientes colecciones contienen artículos importantes sobre sistemas operativos de tiempo real y planificación: [KRIS94], [STAN93], [LEE93] y [TILB91]. [SHA90] proporciona una buena explicación sobre la inversión de prioridad, la herencia de prioridad y el techo de prioridad. [ZEAD97] analiza las prestaciones del planificador de tiempo real del SVR4. [LIND04] proporciona un resumen del planificador de Linux 2.6; [LOVE04] contiene una exposición más detallada.

KRIS94 Krishna, C. y Lee, Y., eds. «Special Issue on Real-Time Systems.» *Proceedings of the IEEE*, Enero 1994.

LEE93 Lee, Y. y Krishna, C., eds. *Readings in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

LIND04 Lindsley, R. «What's New in the 2.6 Scheduler.» *Linux Journal*, Marzo 2004.

LIU00 Liu, J. *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

LOVE04 Love, R. *Linux Kernel Development*. Indianapolis, IN: Sams Publishing, 2004.

SHA90 Sha, L.; Rajkumar, R. y Lehoczky, J. «Priority Inheritance Protocols: An Approach to Real-Time Synchronization.» *IEEE Transactions on Computers*, Septiembre 1990.

STAN93 Stankovic, J. y Ramamritham, K., eds. *Advances in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

TILB91 Tilborg, A. y Koob, G.. eds. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Boston: Kluwer Academic Publishers, 1991.

WEND89 Wendorf, J.; Wendorf, R. y Tokuda, H. «Scheduling Operating System Processing on Small-Scale Microprocessors.» *Proceedings, 22nd Annual Hawaii International Conference on System Science*, Enero 1989.

ZEAD97 Zeadally, S. «An Evaluation of the Real-Time Performance of SVR4.0 and SVR4.2.» *Operating Systems Review*, Enero 1977.

10.8. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

compartición de carga	planificación de pandilla	sistema operativo de tiempo-real
granularidad	planificación de tasa monótona	tarea aperiódica
inversión de prioridad	planificación de tiempo real	tarea de tiempo real duro
inversión de prioridad ilimitada	planificación por plazo	tarea de tiempo real suave
operación de fallo suave	sensibilidad	tarea periódica
planificación de hilos	sistema operativo determinista	

CUESTIONES DE REPASO

10.1. Enumere y defina brevemente cinco categorías diferentes de granularidad de sincronización.

10.2. Enumere y defina brevemente cuatro técnicas de planificación de hilos.

10.3. Enumere y defina brevemente tres versiones de compartición de carga.

10.4. ¿Cuál es la diferencia entre tareas de tiempo real duro y suave?

10.5. ¿Cuál es la diferencia entre tareas de tiempo real periódicas y aperiódicas?

10.6. Enumere y defina brevemente cinco áreas generales de requisitos para un sistema operativo de tiempo real.

10.7. Enumere y defina brevemente cuatro clases de algoritmos de planificación de tiempo real.

10.8. ¿Qué elementos de información sobre una tarea pueden ser útiles en la planificación de tiempo real?

PROBLEMAS

10.1. Considere un conjunto de tres tareas periódicas con los perfiles de ejecución de la Tabla 10.5. Desarrolle diagramas de planificación similares a los de la Figura 10.5 para este conjunto de tareas.

10.2. Considere un conjunto de cinco tareas aperiódicas con los perfiles de ejecución de la Tabla 10.6. Desarrolle diagramas de planificación similares a los de la Figura 10.6 para este conjunto de tareas.

10.3. Este problema demuestra que, aunque la Ecuación (10.2) para la planificación de tasa monótona es una condición suficiente para una planificación satisfactoria, no es una condición necesaria [algunas veces es posible una planificación satisfactoria aunque la Ecuación (10.2) no se satisfaga].

a) Considere el siguiente conjunto de tareas periódicas independientes:

- Tarea P_1: $C_1 = 20$; $T_1 = 100$
- Tarea P_2: $C_2 = 30$; $T_2 = 145$

¿Pueden planificarse satisfactoriamente estas tareas usando el planificador de tasa monótona?

b) Añada ahora la siguiente tarea al conjunto:

- Tarea P_3: $C_3 = 68$; $T_3 = 150$

¿Se satisface la Ecuación (10.2)?

c) Suponga que la primera instancia de las tres tareas precedentes llega en el instante $t = 0$. Asuma que el primer plazo de tiempo para cada tarea es el siguiente:

$$D_1 = 100; D_2 = 145; D_3 = 150$$

Usando planificación de tasa monótona, ¿se cumplirán los tres plazos? ¿Qué hay de los plazos de las futuras repeticiones de cada tarea?

10.4. Dibuje un diagrama similar al de la Figura 10.9b que muestre la secuencia de eventos para este mismo ejemplo usando techo de prioridad.

Tabla 10.5. Perfil de ejecución de para el problema 10.1.

Proceso	Tiempo de llegada	Tiempo de ejecución	Plazo de conclusión
A(1)	0	10	20
A(2)	20	10	40
• • •	• • •	• • •	• • •
B(1)	0	10	50
B(2)	50	10	100
• • •	• • •	• • •	• • •
C(1)	0	15	50
C(2)	50	15	100
• • •	• • •	• • •	• • •

Tabla 10.6. Perfil de ejecución para el problema 10.2.

Proceso	Tiempo de llegada	Tiempo de ejecución	Plazo de comienzo
A	10	20	100
B	20	20	30
C	40	20	60
D	50	20	80
E	60	20	70

PARTE V

ENTRADA/SALIDA Y FICHEROS

Probablemente, el sistema de E/S y el sistema de gestión de ficheros son las partes más intrincadas en el diseño de un sistema operativo. Con respecto a la E/S, el aspecto fundamental es el rendimiento. El sistema de E/S es realmente un factor clave para obtener un buen rendimiento en el sistema. Observando el modo de operación interno de un computador, se puede apreciar que la velocidad del computador continúa incrementándose y, en el caso de que un único procesador no sea todavía suficientemente rápido, las configuraciones SMP proporcionan múltiples procesadores para acelerar el trabajo. Las velocidades de acceso a la memoria interna también están incrementándose, aunque no tan rápidamente como la velocidad de procesador. No obstante, con el uso inteligente de uno, dos o incluso más niveles de cache interna, se está logrando mantener el tiempo de acceso de la memoria principal acorde con la velocidad del procesador. Sin embargo, la E/S supone un reto significativo en el rendimiento, particularmente en el caso del almacenamiento en disco.

Con respecto al sistema de ficheros, el rendimiento es también un aspecto a tener en cuenta. Asimismo, entran también en juego otros requisitos de diseño como la fiabilidad y la seguridad. Desde el punto de vista del usuario, el sistema de ficheros es quizás el aspecto más importante del sistema operativo. El usuario desea un acceso rápido a los ficheros, pero también que se garantice que los ficheros no se corrompen y que están seguros frente a los accesos no autorizados.

GUÍA DE LA QUINTA PARTE

CAPÍTULO 11. GESTIÓN DE E/S Y PLANIFICACIÓN DEL DISCO

El Capítulo 11 comienza con una introducción a los dispositivos de almacenamiento de E/S y a la organización del sistema de E/S dentro del sistema operativo. El capítulo continúa con el estudio de diversas estrategias de gestión de *buffers* para mejorar el rendimiento. El resto del capítulo está dedicado a la E/S de disco. Se analizará el modo en que se pueden planificar múltiples peticiones al disco de manera que se aprovechen las características físicas del acceso al disco para mejorar el tiempo de respuesta. A continuación, se examinará el uso de los vectores de discos para mejorar el rendimiento y la fiabilidad. Por último, se estudiará la cache del disco.

CAPÍTULO 12. GESTIÓN DE FICHEROS

El Capítulo 12 ofrece un estudio de diversos tipos de organizaciones de ficheros y examina los aspectos del sistema operativo que están relacionados con la gestión y el acceso a los ficheros. Se estudia la organización física y lógica de los datos. Se examinan los servicios relacionados con la gestión de ficheros que proporciona un sistema operativo convencional a los usuarios. A continuación, se presentan los mecanismos específicos y las estructuras de datos que forman parte del sistema de gestión de ficheros.

CAPÍTULO 11

Gestión de E/S y planificación del disco

◆ ◆

Probablemente, la E/S es el aspecto más intrincado en el diseño de un sistema operativo. Dado que existe una gran variedad de dispositivos y de aplicaciones de los mismos, es difícil desarrollar una solución general uniforme.

Este capítulo comienza con un breve estudio de los dispositivos de E/S y la organización del sistema de E/S. Estos temas, que generalmente pertenecen al ámbito de la arquitectura de computadores, establecen una base para el estudio de la E/S desde el punto de vista del sistema operativo.

En la siguiente sección se estudian aspectos de diseño del sistema operativo, incluyendo los objetivos de diseño y la manera como se puede organizar el sistema de E/S. A continuación, se examinará el uso de buffers en las operaciones de E/S; uno de los servicios básicos de E/S proporcionados por el sistema operativo es un sistema de gestión de buffers, que mejora el rendimiento general.

Las siguientes secciones del capítulo estarán dedicadas a la E/S del disco magnético. En los sistemas contemporáneos, esta forma de E/S es la más importante, resultando fundamental para el rendimiento tal como lo percibe el usuario. Se comienza desarrollando un modelo del rendimiento de la E/S del disco y, a continuación, se estudian diversas técnicas que pueden utilizarse para mejorar el rendimiento.

Por último, un apéndice de este capítulo resume las características de los dispositivos de almacenamiento secundario, incluyendo el disco magnético y la memoria óptica.

◆ ◆

11.1. DISPOSITIVOS DE E/S

Como se mencionó en el Capítulo 1, los dispositivos externos dedicados a la E/S en un computador se pueden agrupar, a grandes rasgos, en tres categorías:

- **Legibles para el usuario.** Adecuados para la comunicación con el usuario del computador. Algunos ejemplos son las impresoras y terminales de visualización gráfica, que constan de pantalla, teclado y, posiblemente, otros dispositivos como un ratón.
- **Legibles para la máquina.** Adecuados para la comunicación con equipamiento electrónico. Algunos ejemplos son las unidades de discos y de cintas, los sensores, los controladores y los activadores.
- **Comunicación.** Adecuados para la comunicación con dispositivo remotos. Algunos ejemplos son los controladores de una línea digital y los módems.

Hay grandes diferencias entre las distintas categorías e incluso dentro de cada una. Entre las diferencias fundamentales se encuentran las siguientes:

- **Velocidad de transferencia de datos.** Puede haber diferencias de varios órdenes de magnitud entre las velocidades de transferencia de datos. La Figura 11.1 muestra algunos ejemplos.
- **Aplicación.** El uso al que está destinado un dispositivo tiene influencia en el software y en las políticas del sistema operativo y de las herramientas que le dan soporte. Por ejemplo, un disco utilizado para almacenar ficheros requiere el soporte de software de gestión de ficheros. Sin embargo, un disco utilizado como almacenamiento de respaldo para las páginas en un esquema de memoria virtual depende del uso del hardware y el software de memoria virtual. Además, estas dos distintas aplicaciones del disco tienen un impacto sobre los algoritmos de planificación de disco (que se estudiarán más adelante en este capítulo). Como un ejemplo adicional, considere un terminal que puede utilizar un usuario normal o un administrador del

sistema. Estos usos implican diferentes niveles de privilegios y posiblemente prioridades diferentes en el sistema operativo.

- **Complejidad de control.** Una impresora requiere una interfaz de control sencilla. Un disco es mucho más complejo. El efecto de estas diferencias en el sistema operativo se filtra hasta cierto punto por el módulo de E/S que controla el dispositivo, como se estudiará en la próxima sección.
- **Unidad de transferencia.** Los datos pueden transferirse como un flujo de bytes o caracteres (por ejemplo, la E/S de un terminal) o en bloques de mayor tamaño (por ejemplo, la E/S de un disco).
- **Representación de datos.** Los dispositivos utilizan diferentes esquemas de codificación de datos, incluyendo diferencias en el código del carácter y en las convenciones sobre la paridad.
- **Condiciones de error.** La naturaleza de los errores, el modo en que se notifican, sus consecuencias y el rango disponible de respuestas difieren considerablemente de un dispositivo a otro.

Esta diversidad hace difícil lograr un enfoque coherente y uniforme para la E/S, tanto desde el punto de vista del sistema operativo como de los procesos de usuario.

11.2. ORGANIZACIÓN DEL SISTEMA DE E/S

La Sección 1.7 presentó tres técnicas para llevar a cabo la E/S:

- **E/S programada.** El procesador envía un mandato de E/S, a petición de un proceso, a un módulo de E/S; a continuación, ese proceso realiza una espera activa hasta que se complete la operación antes de continuar.

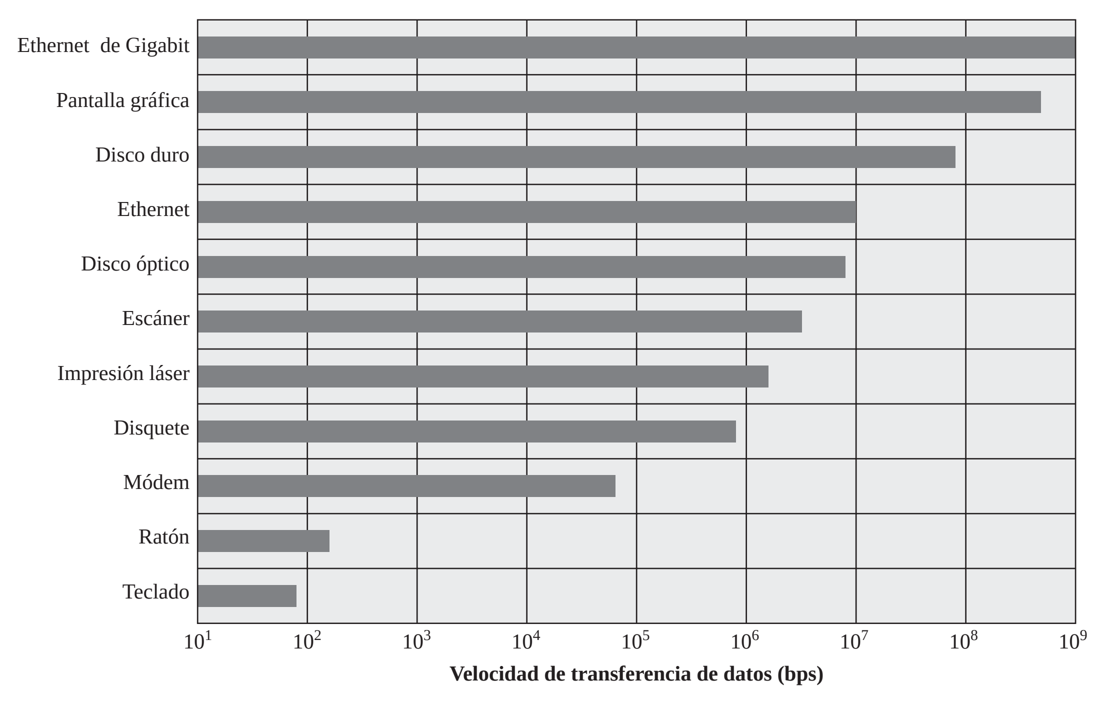


Figura 11.1. Velocidad típica de transferencia de datos de E/S.

- **E/S dirigida por interrupciones.** El procesador emite un mandato de E/S a petición de un proceso y continúa ejecutando las instrucciones siguientes, siendo interrumpido por el módulo de E/S cuando éste ha completado su trabajo. Las siguientes instrucciones pueden ser del mismo proceso, en el caso de que ese proceso no necesite esperar hasta que se complete la E/S. En caso contrario, se suspende el proceso en espera de la interrupción y se realiza otro trabajo.
- **Acceso directo de memoria (*Direct Memory Access*, DMA).** Un módulo de DMA controla el intercambio de datos entre la memoria principal y un módulo de E/S. El procesador manda una petición de transferencia de un bloque de datos al módulo de DMA y resulta interrumpido sólo cuando se haya transferido el bloque completo.

La Tabla 11.1 indica la relación entre estas tres técnicas. En la mayoría de los computadores, el DMA es la forma de transferencia predominante, a la que el sistema operativo debe dar soporte.

Tabla 11.1. Técnicas de E/S.

	Sin interrupciones	Con interrupciones
Transferencia de E/S a memoria a través del procesador	E/S programada	E/S dirigida por interrupciones
Transferencia directa de E/S a memoria		Acceso directo a memoria (DMA)

LA EVOLUCIÓN DEL SISTEMA DE E/S

Según los computadores han ido evolucionando, ha habido una tendencia a una creciente complejidad y sofisticación de los componentes individuales, y esta tendencia en ninguna otra parte es tan evidente como en las funciones de E/S. Las etapas de esta evolución se pueden resumir de la siguiente manera:

1. El procesador controla directamente un dispositivo periférico. Esta situación se presenta en dispositivos simples controlados por un microprocesador.
2. Se añade un controlador o módulo de E/S. El procesador usa E/S programada sin interrupciones. Con este paso, el procesador se independiza de los detalles específicos de las interfaces de los dispositivos externos.
3. Se utiliza la misma configuración que en la etapa anterior, pero empleando interrupciones. El procesador no necesita gastar tiempo esperando a que se realice una operación de E/S, incrementando de esta manera la eficiencia.
4. Al módulo de E/S se le da control directo de la memoria mediante DMA. Con ello, puede mover un bloque de datos a la memoria sin involucrar el procesador, excepto al principio y al final de la transferencia.
5. Se mejora el módulo de E/S para convertirse en un procesador independiente, con un juego de instrucciones especializadas adaptadas a la E/S. La unidad central de procesamiento (CPU) hace que el procesador ejecute un programa de E/S residente en la memoria principal. El procesador de E/S lee y ejecuta estas instrucciones sin la intervención del procesador. Esto permite que el procesador especifique una secuencia de actividades de E/S, siendo interrumpido sólo cuando se termine la secuencia completa.

6. El módulo de E/S tiene su propia memoria local y es, de hecho, un computador por derecho propio. Con esta arquitectura, se pueden controlar un gran conjunto de dispositivos de E/S, con una intervención mínima por parte del procesador. Un uso común de esta arquitectura ha sido controlar la comunicación con terminales interactivos. El procesador de E/S se encarga de la mayoría de las tareas involucradas en el control de los terminales.

Según se avanza a lo largo de este proceso de evolución, cada vez hay una mayor parte de las tareas de E/S que se realizan sin la intervención del procesador. El procesador va quedando relevado gradualmente de estas tareas relacionadas con la E/S, mejorando el rendimiento. En las dos últimas etapas (5 y 6), se produce un cambio principal con la introducción del concepto de un módulo de E/S capaz de ejecutar un programa.

Una nota sobre la terminología: en todos los módulos descritos en las etapas desde la cuarta a la sexta, el uso del término *acceso directo a memoria* (*Direct Access Memory*, DMA) es apropiado, debido a que todos estos tipos de módulos implican el control directo de la memoria principal por parte del módulo de E/S. Asimismo, el módulo de E/S de la quinta etapa se denomina usualmente **canal de E/S**, y el de la sexta etapa **procesador de E/S**; sin embargo, cada término es, en ocasiones, aplicado a ambas situaciones. En la última parte de esta sección, se utilizará el término *canal de E/S* para referirse a ambos tipos de módulos de E/S.

ACCESO DIRECTO A MEMORIA

La Figura 1.12 indica, en términos generales, la lógica del DMA. La unidad de DMA es capaz de imitar al procesador, tomando el control del bus del sistema tal como lo hace un procesador. La unidad de DMA necesita hacerlo para transferir los datos desde y hacia la memoria usando el bus del sistema.

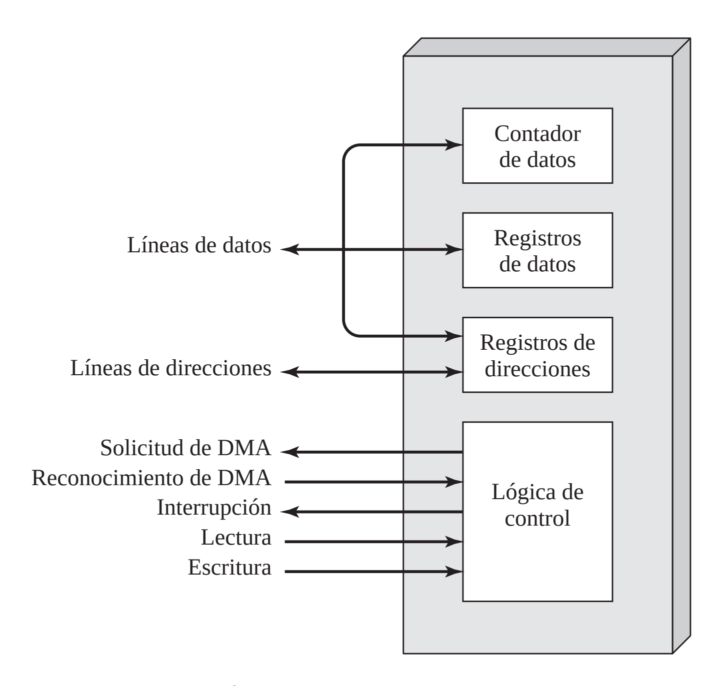


Figura 11.2. Típico diagrama de bloques del DMA.

La técnica del DMA funciona de la manera que se describe a continuación. Cuando el procesador quiere leer o escribir un bloque de datos, envía un mandato al módulo de DMA especificándole la siguiente información:

- Si se trata de una operación de lectura o de escritura, utilizando para ello la línea de control de lectura o escritura que existe entre el procesador y el módulo de DMA.
- La dirección del dispositivo de E/S involucrado, comunicándoselo mediante las líneas de datos.
- La dirección inicial de memoria que se pretende leer o escribir, comunicándoselo mediante las líneas de datos y almacenándose en el registro de dirección del módulo de DMA.
- El número de palabras que se van a leer o escribir, comunicándoselo de nuevo mediante las líneas de datos y almacenándose en el registro contador de datos.

A continuación, el computador continúa con otro trabajo. Ha delegado esta operación de E/S al módulo de DMA. El módulo de DMA transfiere el bloque completo de datos, palabra a palabra, directamente desde la memoria o hacia a ella, sin pasar por el procesador. Cuando se completa la transferencia, el módulo DMA envía una señal de interrupción al procesador. Por tanto, el procesador está involucrado sólo al principio y al final de la transferencia (Figura 1.19c).

El mecanismo de DMA se puede configurar de varias maneras. En la Figura 11.3 se muestran algunas alternativas. En el primer ejemplo, todos los módulos comparten el mismo bus de sistema. El módulo de DMA, actuando como un procesador subordinado, usa E/S programada para intercambiar datos entre la memoria y un módulo de E/S a través del módulo de DMA. Esta configuración, aunque pueda ser económica, es claramente ineficiente: al igual que ocurre con la E/S programada controlada por el procesador, cada transferencia de una palabra consume dos ciclos de bus (petición de transferencia seguida por la transferencia).

El número de ciclos de bus requeridos puede recortarse sustancialmente integrando el DMA y las funciones de E/S. Como indica la Figura 11.3b, esto significa que hay un camino entre el módulo de DMA y uno o más módulos de E/S que no incluye el bus del sistema. La lógica del DMA puede ser realmente parte de un módulo de E/S, o puede ser un módulo separado que controla uno o más módulos de E/S. Este concepto se puede llevar un paso más allá conectando los módulos de E/S al módulo DMA utilizando un bus de E/S (Figura 11.3c). Esto reduce a sólo uno el número de interfaces de E/S en el módulo de DMA y proporciona una configuración fácilmente expansible. En todos estos casos (Figuras 11.3 b y c), el módulo de DMA utiliza el bus del sistema, que comparte con el procesador y la memoria principal, sólo para intercambiar datos con la memoria y señales de control con el procesador. El intercambio de datos entre los módulos de DMA y de E/S tiene lugar fuera del bus del sistema.

11.3. ASPECTOS DE DISEÑO DEL SISTEMA OPERATIVO

OBJETIVOS DE DISEÑO

Hay dos objetivos de suma importancia en el diseño del sistema de E/S: eficiencia y generalidad. La **eficiencia** es importante debido a que las operaciones de E/S usualmente significan un cuello de botella en un computador. Examinando de nuevo la Figura 11.1, se observará que la mayoría de los dispositivos de E/S son extremadamente lentos comparados con la memoria principal y el procesador. Una manera de afrontar este problema es la multiprogramación, que, como ya se estudió previamente, permite que algunos procesos esperen por la finalización de operaciones de E/S mientras se

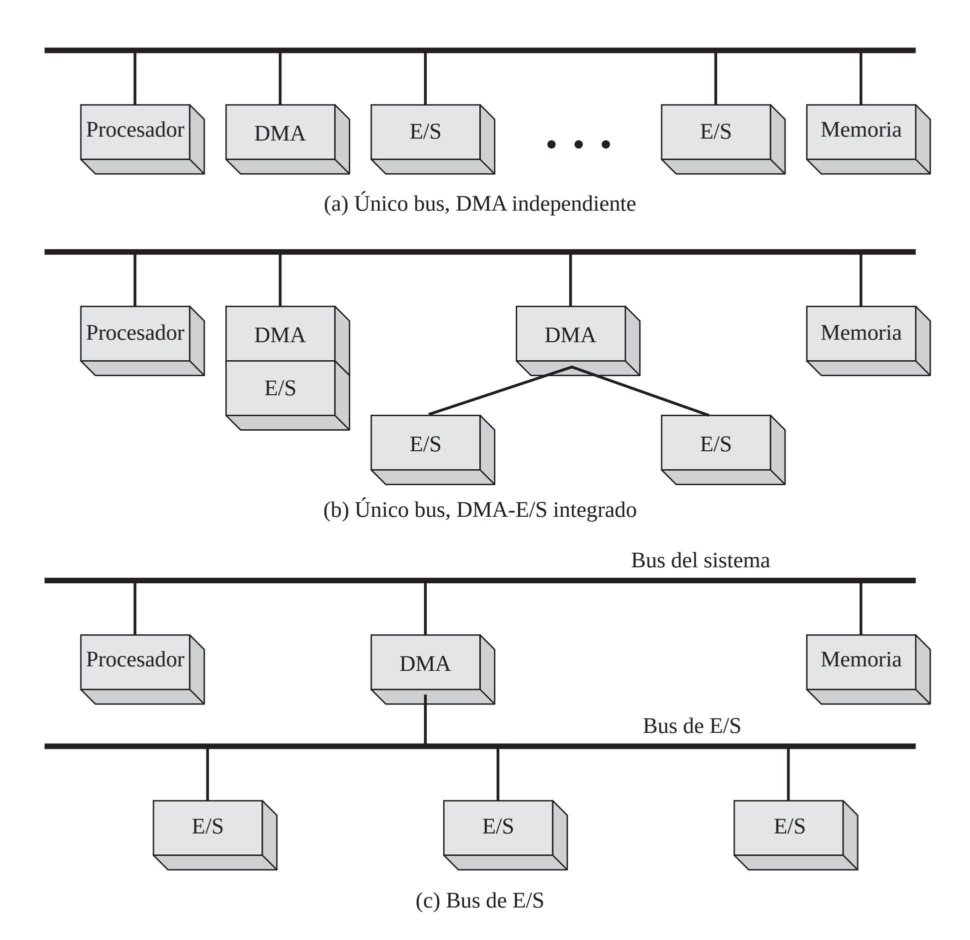


Figura 11.3. Configuraciones de DMA alternativas.

está ejecutando otro proceso. Sin embargo, a pesar del enorme tamaño de la memoria principal de las máquinas de hoy en día, se dará con cierta frecuencia la situación en la que la E/S no puede seguir el ritmo del procesador. Se puede utilizar el intercambio para poder tener más procesos listos de manera que se pueda mantener ocupado al procesador, pero esto en sí mismo es una operación de E/S. Por tanto, se ha dedicado un considerable esfuerzo en el diseño de la E/S para plantear esquemas que mejoren la eficiencia de E/S. El área que ha recibido la mayor atención, por su importancia, es la E/S de disco, por lo que gran parte de este capítulo estará dedicado al estudio de la eficiencia de la E/S de disco.

El otro objetivo principal es la **generalidad.** En aras de la simplicidad y la eliminación de errores, es deseable manejar todos los dispositivos de una manera uniforme. Esta afirmación se aplica tanto al modo en que los procesos ven los dispositivos de E/S como a la manera en que el sistema operativo gestiona los dispositivos y las operaciones de E/S. Debido a la diversidad de las características de los dispositivos, es difícil en la práctica alcanzar una total generalidad. Lo que hay que hacer es utilizar una estrategia modular jerárquica para diseñar las funciones de E/S. Esta estrategia esconde la mayoría de los detalles del dispositivo de E/S en las rutinas de nivel inferior de manera que los procesos de usuario y los niveles más altos del sistema operativo contemplan los dispositivos en términos de funciones generales, tales como lectura, escritura, abrir, cerrar, establecer un cerrojo y liberarlo. A continuación, se estudia esta estrategia.

ESTRUCTURA LÓGICA DEL SISTEMA DE E/S

En el Capítulo 2, en el estudio de la estructura del sistema, se enfatizó la naturaleza jerárquica de los sistemas operativos modernos. La filosofía jerárquica se basa en que las funciones del sistema operativo deberían estar separadas dependiendo de su complejidad, su escala de tiempo característica y su nivel de abstracción. Esta estrategia conduce a una organización del sistema operativo en una serie de niveles. Cada nivel realiza un subconjunto relacionado de las funciones requeridas del sistema operativo y se apoya en el nivel inferior subyacente para realizar funciones más básicas y ocultar los detalles de esas funciones, proporcionando servicios al siguiente nivel superior. Idealmente, los niveles se deberían definir de manera que los cambios en un nivel no requieran cambios en otros niveles. Por tanto, se ha descompuesto un problema en varios subproblemas más manejables.

En general, los niveles inferiores tratan con una escala de tiempo mucho más corta. Algunas partes del sistema operativo deben interaccionar directamente con el hardware del computador, donde los eventos pueden tener una escala de tiempo tan breve como unos pocos nanosegundos. En el otro extremo del espectro, se sitúan las partes del sistema operativo que se comunican con el usuario, que emite mandatos a un ritmo mucho más sosegado, quizás uno cada pocos segundos. El uso de un conjunto de niveles se ajusta perfectamente a este entorno.

Aplicando esta filosofía específicamente al sistema de E/S se llega al tipo de organización sugerida en la Figura 11.4 (compárese con la Tabla 2.4). Los detalles de organización dependerán del tipo de dispositivo y su aplicación. Las tres estructuras lógicas más importantes están representadas en la figura. Por supuesto, un sistema operativo puede que no se ajuste exactamente a estas estructuras. Sin embargo, los principios generales son válidos, por lo que el esquema de E/S de la mayoría de los sistemas operativos encaja aproximadamente en el descrito.

Considérese en primer lugar el caso más simple, el de un dispositivo periférico local que se comunica de una manera sencilla, tal como un flujo de bytes o registros (Figura 11.4a). Los niveles involucrados son los siguientes:

- **E/S lógica.** El módulo de E/S lógica trata a los dispositivos como un recurso lógico y no se ocupa de los detalles del control real del dispositivo. El módulo de E/S lógica se ocupa de la gestión de las tareas generales de E/S para los procesos de usuario, permitiéndolos tratar con el dispositivo en términos de un identificador de dispositivo y con mandatos sencillos como abrir, cerrar, leer y escribir.

- **E/S de dispositivo.** La operaciones requeridas y los datos (caracteres en los *buffers*, registros, etc.) se convierten en las secuencias apropiadas de instrucciones de E/S, mandatos del canal y órdenes del controlador. Se pueden utilizar técnicas de uso de *buffers* para mejorar la utilización.

- **Planificación y control.** La gestión real de la cola y la planificación de las operaciones de E/S se producen en este nivel, así como el control de las operaciones. Por tanto, en este nivel se manejan las interrupciones y se recoge el estado de la E/S y se informa del mismo. Este es el nivel de software que realmente interactúa con el módulo de E/S y, por tanto, con el hardware del dispositivo.

Para un dispositivo de comunicación, la estructura de E/S (Figura 11.4b) se parece mucho a la anteriormente descrita. La diferencia fundamental es que el módulo de E/S lógica se reemplaza por una arquitectura de comunicación, que puede a su vez consistir de varios niveles. Un ejemplo es TCP/IP, que se estudiará en el Capítulo 13.

La Figura 11.4c muestra una estructura representativa para la gestión de E/S en un dispositivo de almacenamiento secundario que proporciona soporte a un sistema de ficheros. Los tres niveles que no se han presentado previamente son:

- **Gestión de directorios.** En este nivel, los nombres simbólicos de los ficheros se convierten en identificadores que o bien hacen referencia directamente al fichero o indirectamente a través de un descriptor de fichero o una tabla de índices. Este nivel también se ocupa de las operaciones de usuario que afectan al directorio de ficheros, tales como añadir, borrar y reorganizar.

- **Sistema de ficheros.** Este nivel trata con la estructura lógica de los ficheros y con las operaciones que pueden especificar los usuarios, como abrir, cerrar, leer y escribir. Los derechos de acceso se gestionan también en este nivel.

- **Organización física.** De la misma manera que las direcciones de memoria virtual deben convertirse en direcciones físicas de memoria principal teniendo en cuenta la estructura de segmentación y paginación, las referencias lógicas a ficheros y registros se deben convertir en direcciones físicas del almacenamiento secundario, teniendo en cuenta la estructura de pistas

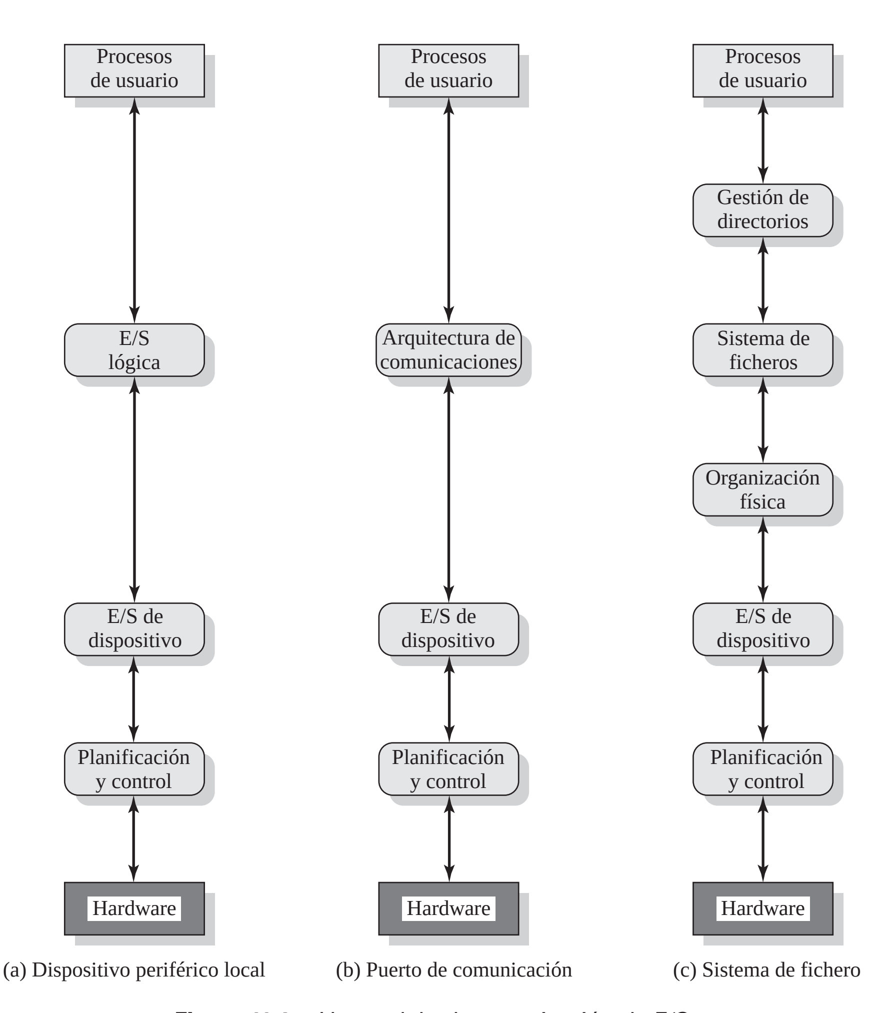


Figura 11.4. Un modelo de organización de E/S.

y sectores del dispositivo de almacenamiento secundario. La asignación de espacio del almacenamiento secundario y de *buffers* del almacenamiento principal se tratan también en este nivel.

Dada la importancia del sistema de ficheros, se dedicará un cierto tiempo en este capítulo y en el próximo a estudiar sus diversos componentes. El estudio de este capítulo se centra en los tres niveles inferiores, mientras que en el Capítulo 12 se presentarán los dos niveles superiores.

11.4. UTILIZACIÓN DE BUFFERS DE E/S

Supóngase que un proceso de usuario desea leer bloques de datos de una cinta de uno en uno, teniendo cada bloque una longitud de 512 bytes. Los datos tienen que leerse en una zona de datos del espacio de direcciones del proceso de usuario que corresponde con las direcciones virtuales desde la 1000 hasta 1511. La manera más sencilla sería enviar un mandato de E/S (algo similar a Leer_Bloque[1000, cinta]) a la unidad de cinta y, a continuación, esperar hasta que los datos estén disponibles. La espera puede ser activa (comprobando continuamente el estado del dispositivo) o, de manera más práctica, suspendiendo el proceso hasta que se produzca una interrupción.

Esta estrategia conlleva dos problemas. Primero, el programa se queda esperando a que se complete la relativamente lenta operación de E/S. El segundo problema es que esta estrategia de E/S interfiere con las decisiones de intercambio del sistema operativo. Las posiciones virtuales desde la 1000 a la 1511 deben permanecer en la memoria principal durante el curso de la transferencia del bloque. En caso contrario, se pueden perder algunos datos. Si se utiliza paginación, al menos la página que contiene las posiciones de memoria requeridas debe fijarse como residente en la memoria principal. Por tanto, aunque algunas páginas del proceso pueden expulsarse al disco, es imposible expulsar completamente al proceso, incluso aunque lo precise el sistema operativo. Nótese también que hay riesgo de que se produzca el interbloqueo de un único proceso. Si un proceso emite un mandato de E/S, se suspende esperando el resultado, y, a continuación, se expulsa antes del inicio de la operación, el proceso se bloquea esperando el evento de E/S y la operación de E/S se bloquea esperando a que el proceso se traiga de nuevo a memoria. Para evitar este interbloqueo, la memoria de usuario involucrada en la operación de E/S debe quedarse residente en la memoria principal inmediatamente antes de que se emita la petición de E/S, incluso aunque la operación de E/S se encole y no pueda ejecutarse durante algún tiempo.

Las mismas consideraciones se aplican a una operación de salida. Si se está transfiriendo un bloque desde un área del proceso de usuario directamente a un módulo de E/S, el proceso se bloquea durante la transferencia y el proceso no puede expulsarse.

Para evitar estas sobrecargas e ineficiencias, es a veces conveniente realizar las transferencias de entrada antes de que se hagan las peticiones correspondientes y llevar a cabo las transferencias de salida un cierto tiempo después de que se haya hecho la petición. Esta técnica se conoce como E/S con *buffers*. En esta sección, se analizarán algunos esquemas de gestión de *buffers* que proporcionan los sistemas operativos para mejorar el rendimiento del sistema.

A la hora de analizar las distintas estrategias de gestión de *buffers*, es a veces importante hacer una distinción entre dos tipos de dispositivos de E/S: orientados a bloques y orientados a flujo de caracteres. Un dispositivo **orientado a bloques** almacena información en bloques que son usualmente de tamaño fijo realizándose las transferencias de bloque en bloque. Generalmente, es posible hacer referencia a los datos mediante su número de bloque. Los discos y las cintas son ejemplos de dispositivos orientados a bloques. Un dispositivo **orientado a flujo de caracteres** transfiere los datos, tanto de entrada como de salida, como un flujo de bytes, sin estructura de bloques. Los terminales, las impresoras, los puertos de comunicación, el ratón y otros dispositivos apuntadores, y

la mayoría de los dispositivos que no son de almacenamiento secundario están orientados a flujos de caracteres.

BUFFER ÚNICO

El tipo más sencillo de esquema que puede proporcionar el sistema operativo es el *buffer* único (Figura 11.5b). Cuando un proceso de usuario emite una petición de E/S, el sistema operativo asigna un *buffer* para la operación en la parte de sistema de la memoria principal.

Para dispositivos orientados a bloques, el esquema con un *buffer* único se puede describir de la siguiente manera: las transferencias de entrada usan el *buffer* del sistema. Cuando se completa la transferencia, el proceso mueve el bloque al espacio de usuario e inmediatamente pide otro bloque. A esto se le denomina lectura adelantada, o entrada anticipada; esta operación se realiza con la esperanza de que se acabará necesitando ese bloque. Para muchos tipos de cómputos, se trata de una suposición razonable durante la mayor parte del tiempo porque normalmente se accede a los datos de forma secuencial. Sólo al final de una secuencia de procesamiento se leerá innecesariamente un bloque.

Esta estrategia proporcionará generalmente una cierta mejora en el rendimiento comparada con la alternativa de no usar *buffers* en el sistema. El proceso de usuario puede estar procesando un bloque de datos mientras se está leyendo el bloque siguiente. El sistema operativo es capaz de expulsar

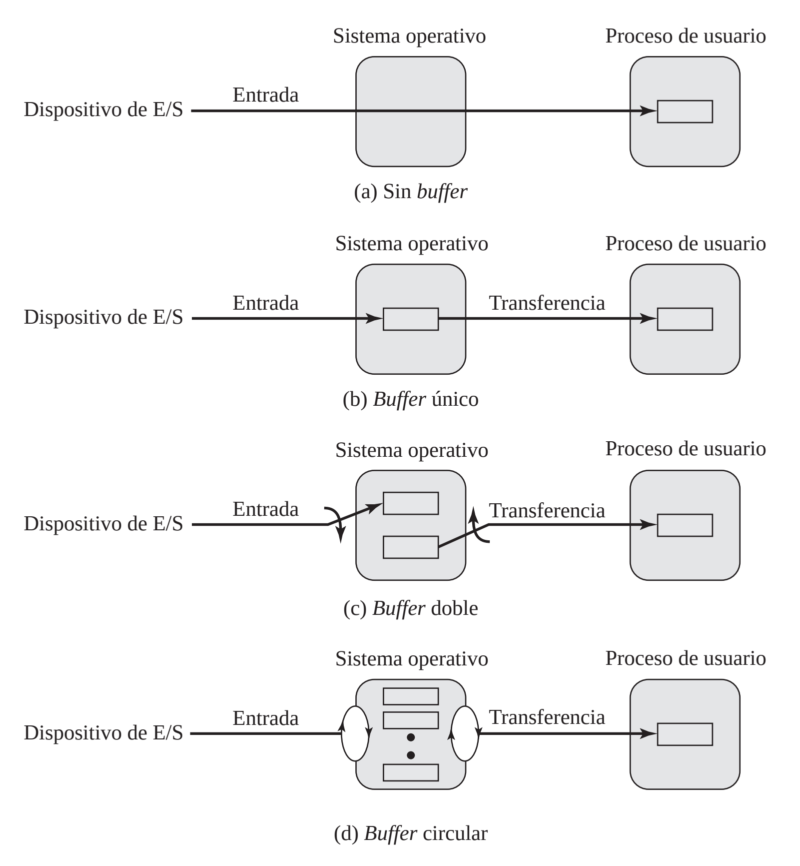


Figura 11.5. Esquemas de uso de *buffers* de E/S (entrada/salida).

al proceso puesto que se está llevando a cabo la operación de entrada en la memoria del sistema en vez de en la memoria del proceso de usuario. Esta técnica, sin embargo, complica la lógica en el sistema operativo. El sistema operativo debe hacer un seguimiento de la asignación de *buffers* del sistema a los procesos de usuario. También afecta a la lógica del intercambio: si la operación de E/S afecta al mismo disco que se está usando para el intercambio, prácticamente no tiene sentido encolar las escrituras en disco requeridas para expulsar el proceso sobre ese mismo dispositivo. Este intento de expulsar el proceso y liberar memoria principal no comenzará en sí mismo hasta que termine la operación de E/S previa, en cuyo momento la expulsión del proceso al disco puede haber dejado de ser apropiada.

Se pueden aplicar consideraciones similares a la salida orientada a bloques. Cuando se transfieren los datos a un dispositivo, en primer lugar, se copian desde el espacio de usuario hasta el *buffer* del sistema, desde donde se escribirán definitivamente. El proceso solicitante puede en este momento continuar o ser expulsado cuando sea necesario.

[KNUT97] sugiere una comparación de rendimiento, un tanto tosca pero bastante informativa, entre usar un *buffer* único y no usar ninguno. Supóngase que T es el tiempo requerido para leer un bloque y que P es el tiempo de proceso que transcurre entre sucesivas peticiones de lectura. Sin *buffer,* el tiempo de ejecución por bloque es básicamente $T + P$. Con un *buffer* único, el tiempo es el máximo de $[P, T] + M$, donde M es el tiempo requerido para mover los datos desde el *buffer* del sistema a la memoria de usuario. En la mayoría de los casos, el tiempo de ejecución por bloque es sustancialmente menor usando un *buffer* único que no usando ninguno.

En el caso de la E/S orientada a flujo de caracteres, el esquema de *buffer* único se puede utilizar en un modo de operación línea a línea o en uno byte a byte. El modo de operación línea a línea es apropiado para los terminales en modo texto, en los que el texto se va desplazando verticalmente (a veces denominados terminales no inteligentes). Con este tipo de terminal, la entrada del usuario en el terminal es línea a línea, usando un retorno de carro para indicar el final de una línea, y, de manera similar, la salida es también línea a línea. Una impresora de líneas es otro ejemplo de este tipo de dispositivos. La operación byte a byte se utiliza en los terminales en modo formulario, donde cada tecla pulsada es significativa, y para muchos otros periféricos, como los sensores y los controladores.

En el caso de E/S línea a línea, se puede utilizar el *buffer* para almacenar una única línea. El proceso de usuario se suspende durante la entrada de datos, esperando la llegada de una línea completa. Con respecto a la salida, el proceso de usuario puede copiar una línea de salida en el *buffer* y continuar el procesamiento. No se necesita suspender el proceso a menos que tenga una segunda línea de salida que enviar al *buffer* antes de que se vacíe el mismo al terminar la primera operación de salida. En el caso de E/S byte a byte, la interacción entre el sistema operativo y el proceso de usuario sigue el modelo productor-consumidor estudiado en el Capítulo 5.

BUFFER DOBLE

Se puede hacer una mejora sobre la técnica del *buffer* único asignando a la operación dos *buffers* del sistema (Figura 11.5c). Con este nuevo esquema, un proceso transfiere datos a (desde) un *buffer* mientras el sistema operativo vacía (o llena) el otro. Esta técnica se conoce como ***buffer* doble** o **intercambio de *buffers***.

En cuanto a la transferencia orientada a bloques, se puede estimar aproximadamente el tiempo de ejecución como el máximo de $[P, T]$. Por tanto, es posible mantener el dispositivo orientado a bloques trabajando a toda velocidad si $P \leq T$. Por otro lado, si $P > T$, el *buffer* doble asegura que el proceso no tendrá que esperar la finalización de la E/S. En cualquier caso, se alcanza una mejora con respecto al uso de un *buffer* único. De nuevo, esta mejora conlleva un incremento de la complejidad.

Con respecto a la entrada orientada a flujo de caracteres, se plantean de nuevo los dos modos de operación alternativos. En el caso de la E/S línea a línea, el proceso de usuario no necesita suspenderse debido a una operación de entrada o salida, a menos que el proceso vaya por delante del *buffer* doble. En el caso de operación byte a byte, el *buffer* doble no ofrece ninguna ventaja adicional sobre un *buffer* único de doble longitud. En ambos casos, se utiliza el modelo productor-consumidor.

BUFFER **CIRCULAR**

Un esquema de *buffer* doble debería suavizar el flujo de datos entre un dispositivo de E/S y un proceso. Si el interés está centrado en el rendimiento de un determinado proceso, se desearía que la operación de E/S fuera capaz de mantener el ritmo del proceso. El *buffer* doble puede ser inadecuado si el proceso realiza ráfagas rápidas de E/S. En este caso, el problema puede aliviarse frecuentemente utilizando más de dos *buffers*.

Cuando se utilizan más de dos *buffers*, al conjunto de *buffers* se le denomina *buffer* circular (Figura 11.5d), siendo cada *buffer* individual una unidad del *buffer* circular. Se trata simplemente de un modelo productor-consumidor con un *buffer* acotado, como se estudió en el Capítulo 5.

LA UTILIDAD DEL USO DE ***BUFFERS***

El uso de *buffers* es una técnica que amortigua los picos en la demanda de E/S. Sin embargo, por muchos *buffers* que se utilicen, estos no permitirán a un dispositivo de E/S mantener el ritmo de un proceso indefinidamente cuando la demanda media del proceso sea mayor que la que puede servir el dispositivo de E/S. Incluso con múltiples *buffers*, todos los *buffers* acabarán llenándose y el proceso tendrá que esperar después de procesar cada fragmento de datos. Sin embargo, en un entorno de multiprogramación, donde hay diversas actividades de E/S y distintos procesos que hay que atender, el uso de *buffers* es una técnica que puede incrementar la eficiencia del sistema operativo y el rendimiento de los procesos individuales.

11.5. PLANIFICACIÓN DEL DISCO

Durante los últimos cuarenta años, el incremento de la velocidad de los procesadores y de la memoria principal ha sobrepasado con creces el de la velocidad de acceso al disco, incrementándose en aproximadamente dos órdenes de magnitud mientras que la velocidad del disco lo ha hecho en un orden de magnitud. El resultado es que los discos son actualmente al menos cuatro órdenes de magnitud más lentos que la memoria principal. Además, se espera que esta diferencia aumente en el futuro inmediato. Por tanto, el rendimiento del subsistema de almacenamiento de disco es de vital importancia, por lo que se han realizado muchas investigaciones en esquemas para mejorar ese rendimiento. En esta sección, se resaltan algunos de los aspectos fundamentales y se revisan las técnicas más importantes. Dado que el rendimiento del sistema de discos está estrechamente asociado a aspectos de diseño del sistema de ficheros, este estudio continuará en el Capítulo 12.

PARÁMETROS DE RENDIMIENTO DEL DISCO

Los detalles reales de la operación de E/S del disco dependen del computador, del sistema operativo, y de la naturaleza del hardware del canal de E/S y del controlador del disco. En la Figura 11.6 se muestra un diagrama general de tiempos de una transferencia de E/S de disco.

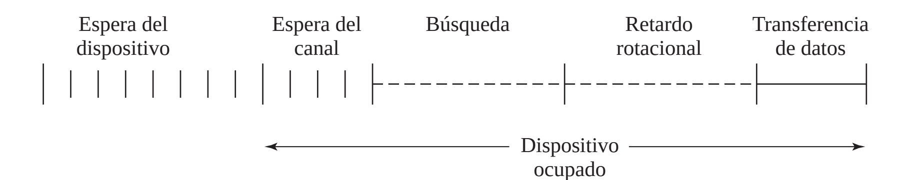


Figura 11.6. Diagrama de tiempos de una transferencia de E/S de disco.

Cuando está en funcionamiento la unidad de disco, el disco rota a una velocidad constante. Para leer o escribir, la cabeza se debe posicionar en la pista deseada y en el principio del sector requerido de dicha pista[1]. La selección de pista implica el movimiento de la cabeza en un sistema de cabeza móvil o la selección de manera electrónica de una cabeza en un sistema de cabeza fija. En un sistema de cabeza móvil, el tiempo que se tarda en situar la cabeza en la pista se denomina **tiempo de búsqueda**. En cualquier caso, una vez que se selecciona la pista, el controlador del disco espera hasta que el sector apropiado rote debajo de la cabeza. El tiempo que tarda en llegar el comienzo del sector hasta debajo de la cabeza se conoce como **retardo rotacional**, o latencia rotacional. La suma del tiempo de búsqueda, en el caso de que lo haya, y el retardo rotacional dan lugar al **tiempo de acceso**, que es el tiempo que se tarda en llegar a estar en posición para realizar la lectura o escritura. Una vez que la cabeza está en posición, la operación de lectura o escritura se realiza cuando el sector se mueve debajo de la cabeza; ésta es la parte de la transferencia de datos de la operación; el tiempo requerido para la transferencia es el **tiempo de transferencia**.

Además del tiempo de acceso y de transferencia, hay varios retardos por esperas en colas normalmente asociadas con una operación de E/S del disco. Cuando un proceso emite una petición de E/S, debe esperar, en primer lugar, en una cola del dispositivo hasta que esté disponible. En ese momento, se asigna el dispositivo al proceso. Si el dispositivo comparte un único canal de E/S o un conjunto de canales de E/S con otras unidades de disco, puede haber una espera adicional hasta que el canal esté disponible. En ese punto, se realiza la búsqueda para comenzar el acceso al disco.

En algunos sistemas de tipo *mainframe*, se utiliza una técnica conocida como detección de posición rotacional (*Rotational Position Sensing*, RPS). Este mecanismo funciona de la forma siguiente: cuando se ha generado el mandato de búsqueda, se libera el canal para manejar otras operaciones de E/S. Cuando se completa la búsqueda, el dispositivo determina cuándo los datos rotan debajo de la cabeza. Cuando ese sector se aproxima a la cabeza, el dispositivo intenta restablecer el camino de comunicación de vuelta a la máquina. Si la unidad de control o el canal están ocupados con otra operación de E/S, el intento de volver a conectarse falla y el dispositivo debe rotar una revolución completa antes de que pueda intentar volverse a conectar, a lo que se denomina un fallo de RPS. Esta situación representa un elemento de retardo adicional que debe añadirse a la línea de tiempo de la Figura 11.6.

Tiempo de búsqueda

El tiempo de búsqueda es el tiempo requerido para mover el brazo del disco a la pista requerida. El tiempo de búsqueda consta de dos componentes fundamentales: el tiempo de arranque inicial y el tiempo que se tarda en atravesar las pistas que tienen que cruzarse una vez que el brazo de acceso

---

[1] *Véase* el Apéndice 11A para un estudio de la organización y el formato del disco.

empieza a moverse. Por desgracia, el tiempo para atravesar las pistas no es una función lineal del número de pistas sino que incluye un tiempo de establecimiento (el tiempo que transcurre desde que se posiciona la cabeza sobre la pista prevista hasta que se confirma su identificación).

Se han producido muchas mejoras gracias al desarrollo de componentes de disco más pequeños y ligeros. Hace algunos años, un disco normal tenía un diámetro de 14 pulgadas (36 cm.), mientras que el tamaño más común actualmente es de 3,5 pulgadas (8,9 cm.), reduciendo la distancia que tiene que moverse el brazo. Un tiempo normal de búsqueda medio en los discos duros actuales está por debajo de los 10 ms.

Retardo rotacional

Los discos, excepto los flexibles, rotan a velocidades que van desde las 3600 rpm (para dispositivos portátiles como cámaras digitales) hasta 15.000 rpm, en el momento en el que se escribió este libro; a esta última velocidad se produce una revolución cada 4 ms. Por tanto, en promedio, el retardo rotacional será de 2 ms. Los discos flexibles normalmente giran a una velocidad entre 300 y 600 rpm. Por tanto, el retardo medio estará entre 100 y 50 ms.

Tiempo de transferencia

El tiempo de transferencia de un disco depende de la velocidad de rotación de acuerdo con la siguiente expresión:

$$T = \frac{b}{rN}$$

donde

T = tiempo de transferencia

b = número de bytes que se van a transferir

N = número de bytes en una pista

r = velocidad de rotación, en revoluciones por segundo

Por tanto, el tiempo de acceso total medio se puede expresar de la siguiente manera:

$$T_a = T_b + \frac{1}{2r} + \frac{b}{rN}$$

donde T_b es el tiempo de búsqueda medio.

Una comparación de tiempos

Con los parámetros previamente definidos, se examinarán dos operaciones de E/S diferentes para mostrar el peligro de confiar en los valores medios. Considere un disco con un tiempo de búsqueda medio de 4ms. según su fabricante, una velocidad de rotación de 7.500 rpm y sectores de 512 bytes con 500 sectores por pista. Supóngase que se quiere leer un fichero que consta de 2.500 sectores, lo que significa un total de 1,28 Mbytes. Se pretende estimar el tiempo total de transferencia.

En primer lugar, supóngase que el fichero se almacena de la forma más compacta posible en el disco. Es decir, el fichero ocupa todos los sectores de 5 pistas adyacentes (5 pistas × 500 sectores/pista = 2.500 sectores). Esto se conoce como *organización secuencial*. El tiempo para leer la primera pista es el siguiente:

Búsqueda media	4	ms
Retardo rotacional	4	ms
Lectura de 500 sectores	8	ms
	16	ms

Supóngase que las restantes pistas se pueden leer ahora sin básicamente tiempo de búsqueda. Es decir, la operación de E/S puede seguir el flujo del disco. Entonces, como mucho, se necesita tener en cuenta el retardo rotacional para cada pista sucesiva. De este modo, se lee cada pista sucesiva en 4 + 8 = 12 ms. Para leer el fichero entero sería necesario:

$$\text{Tiempo total} = 16 + 4 \times 12 = 64 \text{ ms} = 0{,}064 \text{ segundos}$$

A continuación, se calculará el tiempo requerido para leer los mismos datos utilizando un acceso aleatorio en lugar de un acceso secuencial; es decir, los accesos a los sectores se distribuyen de manera aleatoria en el disco. Por cada sector, se tiene:

Búsqueda media	4	ms
Retardo rotacional	4	ms
Lectura de 1 sector	0,016	ms
	8,016	ms

$$\text{Tiempo total} = 2500 \times 8{,}016 = 20.040 \text{ ms} = 20{,}04 \text{ segundos}$$

Es evidente que el orden en el que se leen los sectores del disco tiene un efecto tremendo en el rendimiento de la E/S. En el caso del acceso a un fichero en el que se leen o se escriben múltiples sectores, se tiene algún control sobre el modo en el que se distribuyen los sectores de datos en el disco, como se estudiará en el próximo capítulo. Sin embargo, incluso en el caso del acceso a un fichero, en un entorno de multiprogramación, habrá diversas peticiones de E/S compitiendo por el mismo disco. Por tanto, es importante examinar de qué maneras se puede mejorar el rendimiento de la E/S del disco con respecto al logrado con un acceso al disco puramente aleatorio.

POLÍTICAS DE PLANIFICACIÓN DEL DISCO

En el ejemplo que se acaba de describir, la diferencia en el rendimiento se debe al tiempo de búsqueda. Si las peticiones de acceso a los sectores involucran una selección aleatoria de pistas, el rendimiento del sistema de E/S del disco será el peor posible. Para mejorar el asunto, se necesita reducir el tiempo medio gastado en las búsquedas.

Considere la situación normal en un entorno de multiprogramación, en el que el sistema operativo mantiene una cola de peticiones por cada dispositivo de E/S. Así, en el caso de un único disco, habrá varias peticiones de E/S (lecturas y escrituras) de diversos procesos en la cola. Si se seleccionan elementos de la cola en un orden aleatorio, se puede predecir que las pistas se van a visitar aleatoriamente, proporcionando un rendimiento deficiente. Esta **planificación aleatoria** es útil como un punto de referencia para evaluar otras técnicas.

La Figura 11.7 compara el rendimiento de varios algoritmos de planificación usando un ejemplo de una secuencia de peticiones de E/S. El eje vertical corresponde con las pistas en el disco. El eje horizontal corresponde con el tiempo o, de manera equivalente, el número de pistas atravesadas. En esta figura, se asume que la cabeza del disco está situada inicialmente en la pista 100. En este ejemplo, se supone un disco con 200 pistas y que la cola de peticiones del disco incluye peticiones aleatorias. Las pistas solicitadas, en el orden recibido por el planificador del disco, son 55, 58, 39, 18, 90, 160, 150, 38 y 184. La Tabla 11.2a muestra los resultados en forma de tabla.

Tabla 11.2. Comparación de algoritmos de planificación de disco.

(a) FIFO (comenzando en la pista 100)		(b) SSTF (comenzando en la pista 100)		(c) SCAN (comenzando en la pista 100, en la dirección de números de pista crecientes)		(d) C-SCAN (comenzando en la pista 100, en la dirección de números de pista crecientes)	
Próxima pista accedida	**Número de pistas atravesadas**	**Próxima pista accedida**	**Número de pistas atravesadas**	**Próxima pista accedida**	**Número de pistas atravesadas**	**Próxima pista accedida**	**Número de pistas atravesadas**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
longitud media de búsqueda	55,3	**longitud media de búsqueda**	27,5	**longitud media de búsqueda**	27,8	**longitud media de búsqueda**	35,8

Primero en entrar, primero en salir

La forma más sencilla de planificación corresponde con el algoritmo del primero en entrar, primero en salir (*First In-First Out*, FIFO), que procesa los elementos de la cola en orden secuencial. Esta estrategia tiene la ventaja de ser equitativa, porque toda petición se acaba sirviendo y, además, las peticiones se sirven en el orden recibido. La Figura 11.7a muestra el movimiento del brazo del disco con el algoritmo FIFO.

Con esta planificación, si sólo hay unos pocos procesos que requieren acceso al disco y una gran parte de las peticiones corresponden con sectores agrupados de ficheros, se puede prever un buen rendimiento. Sin embargo, esta técnica tendrá con frecuencia resultados similares a la planificación aleatoria en cuanto al rendimiento, en el caso de que haya muchos procesos compitiendo por el disco. Por tanto, puede ser beneficioso considerar una política de planificación más sofisticada. En la Tabla 11.3 se enumeran varias políticas de planificación, que se analizarán a continuación.

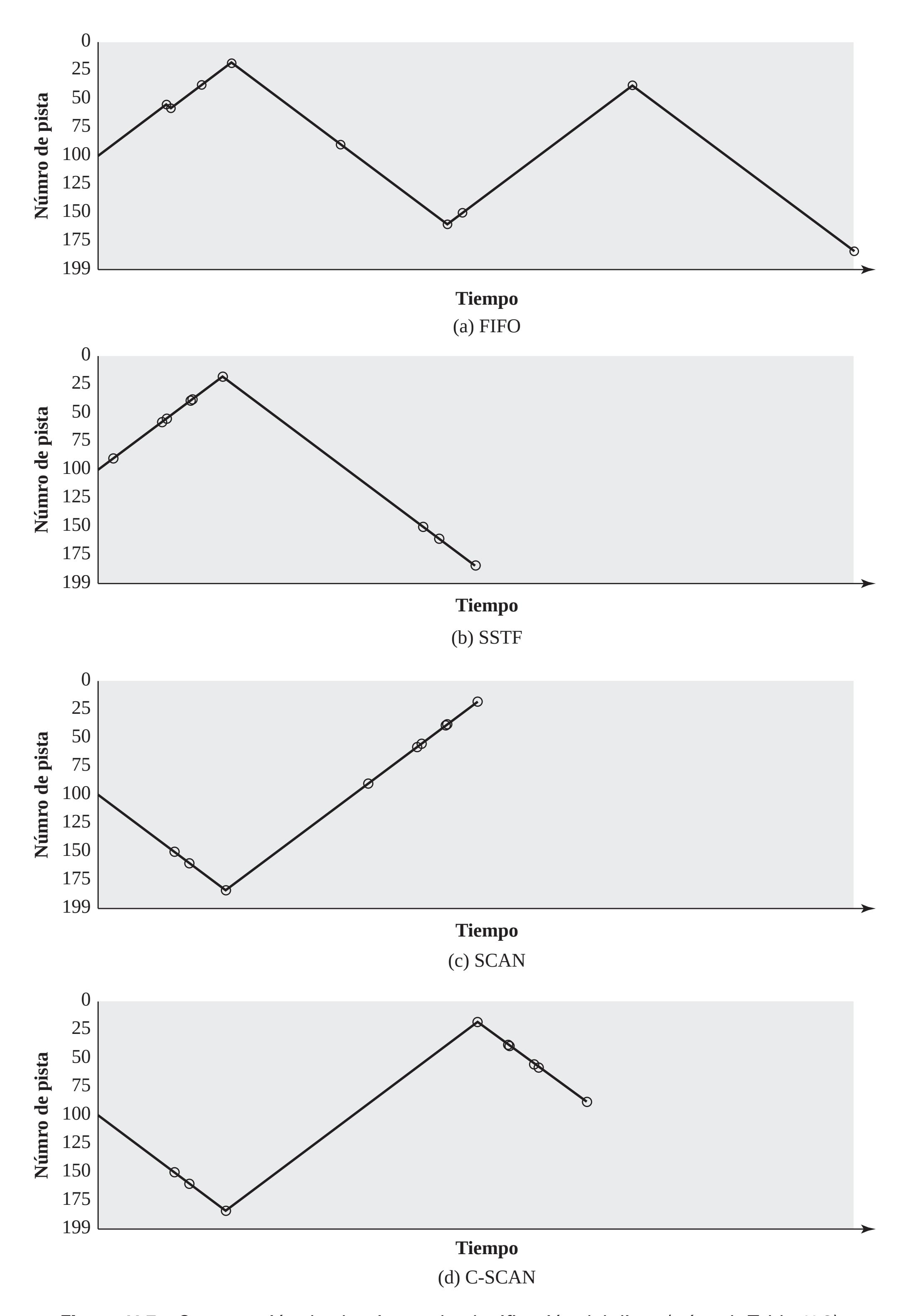


Figura 11.7. Comparación de algoritmos de planificación del disco (*véase* la Tabla 11.3).

Tabla 11.3. Algoritmos de planificación del disco.

Nombre	Descripción	Comentarios
	Selección según el solicitante	
PA	Planificación aleatoria	Para análisis y simulación
FIFO	Primero en entrar, primero en salir	El más equitativo de todos
PRI	Prioridad por proceso	Control externo a la gestión de la cola de disco
LIFO	Último en entrar, primero en salir	Maximiza proximidad y uso de recursos
	Selección según el elemento solicitado	
SSTF	Primero el de tiempo de servicio más corto	Buena utilización; colas pequeñas
SCAN	Recorrido del disco bidireccional	Mejor distribución del servicio
C-SCAN	Recorrido unidireccional con retorno rápido	Menor variabilidad de servicio
SCAN-de-N-pasos	SCAN de *N* registros cada vez	Garantía de servicio
FSCAN	SCAN-de-*N*-pasos con *N* = tamaño de la cola al principio del ciclo de SCAN	Sensible a la carga

Prioridad

En un sistema basado en la prioridad (PRI), la planificación está fuera del control del software de gestión de disco. Esta estrategia no está diseñada para optimizar la utilización del disco sino para satisfacer otros objetivos dentro del sistema operativo. Con frecuencia se les da mayor prioridad a los trabajos por lotes de corta duración y a los trabajos interactivos que a los trabajos más largos que requieren un procesamiento más prolongado. Esto permite que muchos trabajos cortos salgan del sistema rápidamente, pudiendo proporcionar un tiempo de respuesta interactiva adecuado. Sin embargo, los trabajos más largos pueden tener que esperar un tiempo excesivamente largo. Además, una política de este tipo podría llevar a que los usuarios tomaran medidas en su contra, dividiendo sus trabajos en partes más pequeñas para engañar al sistema. Este tipo de política tiende a ser inadecuada para los sistemas de base de datos.

Último en entrar, primero en salir

Sorprendentemente, la política de seleccionar siempre la petición más reciente tiene cierto mérito. En sistemas de procesamiento de transacciones, asignar el dispositivo al usuario más reciente debería producir poco o ningún movimiento del brazo durante un desplazamiento a lo largo de un fichero secuencial. Aprovechar la proximidad mejora el rendimiento y reduce la longitud de la cola. Mientras un trabajo utilice activamente el sistema de ficheros, se procesará tan rápido como sea posible. Sin embargo, si el disco se mantiene ocupado debido a una gran carga de trabajo, hay una evidente posibilidad de inanición. Una vez que un proceso ha insertado una petición de E/S en la cola y pierde su posición en la parte delantera de la misma, el proceso no podrá recuperar dicha posición hasta que se terminen las peticiones que hay delante en la cola.

Las planificaciones FIFO, por prioridad y LIFO (último en entrar, primero en salir) están basadas solamente en atributos de la cola o del demandante. Si el planificador conoce la posición de pista actual, se puede emplear una planificación basada en el elemento solicitado. A continuación se examinan estas políticas.

Primero el de tiempo de servicio más corto

La política SSTF (*Shortest Service Time First*, primero el de tiempo de servicio más corto) consiste en seleccionar la petición de E/S del disco que requiera un menor movimiento del brazo desde su posición actual. De ese modo, siempre se realiza una selección de manera que se produzca un tiempo de búsqueda mínimo. Evidentemente, seleccionar siempre el tiempo de búsqueda mínimo no garantiza que sea mínimo el tiempo de búsqueda medio correspondiente a varios movimientos del brazo. Sin embargo, este esquema debería proporcionar un rendimiento mejor que el algoritmo FIFO. Dado que el brazo puede moverse en dos direcciones, se puede utilizar un algoritmo aleatorio para resolver los casos de empate debido a la existencia de distancias iguales.

La Figura 11.7b y la Tabla 11.2b muestran el rendimiento de SSFT en el mismo ejemplo que se usó con la planificación FIFO.

Scan

Con la excepción de la planificación FIFO, todas las políticas descritas hasta ahora pueden dejar alguna petición sin servir hasta que se vacíe completamente la cola de peticiones. Es decir, podrían llegar continuamente nuevas peticiones que se seleccionarían antes de una petición existente. Una alternativa sencilla que impide este tipo de inanición es el algoritmo SCAN, también conocido como el algoritmo del ascensor porque funciona de manera muy similar a como lo hace un ascensor.

Con el algoritmo SCAN, el brazo sólo debe moverse en una dirección, satisfaciendo todas las peticiones pendientes que encuentre en su camino, hasta que alcanza la última pista en esa dirección o hasta que no haya más peticiones en esa dirección. A esta última mejora se le denomina política LOOK. En ese momento, la dirección de servicio se invierte y la búsqueda continúa en la dirección opuesta, sirviendo de nuevo todas las peticiones en orden.

La Figura 11.7c y la Tabla 11.2c muestran la política SCAN. Como se puede observar, la política SCAN se comporta prácticamente igual que la política SSTF. De hecho, si se hubiera supuesto que el brazo se estaba moviendo en la dirección de las pistas con números más bajos al comienzo del ejemplo, el patrón de planificación habría sido el mismo para SSTF y SCAN. Sin embargo, se trata de un ejemplo estático en el que no se añade ningún elemento nuevo a la cola. En cualquier caso, incluso si la cola está cambiando dinámicamente, SCAN será similar a SSTF a menos que el patrón de peticiones sea inusual.

Nótese que la política SCAN no favorece en su tratamiento al área del disco que se ha atravesado más recientemente. Por tanto, no aprovecha la proximidad tan bien como lo hacen SSTF o incluso LIFO.

No es difícil observar que la política SCAN favorece a los trabajos cuyas peticiones corresponden con las pistas más cercanas tanto a las pistas interiores como a las exteriores, así como a los trabajos que han llegado más recientemente. El primer problema se puede evitar mediante la política C-SCAN, mientras que el segundo problema lo soluciona la política SCAN-de-*N*-pasos.

C-Scan

La política C-SCAN (SCAN circular) restringe la búsqueda a una sola dirección. Por tanto, después de visitar la última pista en una dirección, el brazo vuelve al extremo opuesto del disco y la búsqueda comienza de nuevo. Esto reduce el retardo máximo que pueden experimentar las nuevas peticiones. Con SCAN, si el tiempo previsto para una búsqueda de la pista más interna a la más externa es t, el intervalo de servicio previsto para los sectores en la periferia es $2t$. Con C-SCAN, el intervalo es del orden de $t + s_{máx}$, siendo $s_{máx}$ el tiempo de búsqueda máximo.

La Figura 11.7d y la Tabla 11.2d muestran el comportamiento de C-SCAN.

SCAN-de-*N*-pasos y FSCAN

Con SSFT, SCAN y C-SCAN, es posible que el brazo no pueda moverse durante un periodo de tiempo considerable. Por ejemplo, si uno o más procesos tienen elevadas tasas de acceso a una determinada pista, pueden monopolizar el dispositivo entero debido a sucesivas peticiones a esa pista. Los discos de alta densidad con múltiples superficies se van a ver probablemente más afectados por esta característica que los discos de densidad más baja y/o los discos con solo una o dos superficies. Para evitar este «estancamiento del brazo», la cola de peticiones del disco puede dividirse en segmentos, tal que en cada momento se está procesando un segmento hasta que se complete. Los algoritmos SCAN-de-*N*-pasos y FSCAN son dos ejemplos de esta estrategia.

La política de SCAN-de-*N*-pasos divide la cola de peticiones del disco en varias colas de longitud *N*. En cada momento se procesa solo una cola, utilizando SCAN. Mientras se está procesando una cola, las nuevas peticiones se deben añadir a otra cola. Si hay menos de *N* peticiones disponibles al final de una búsqueda, todas ellas se procesan en la próxima búsqueda. Para valores grandes de *N*, el rendimiento de SCAN-de-*N*-pasos se aproxima al de SCAN; para un valor de $N = 1$, se convierte en una política FIFO.

FSCAN es una política que utiliza dos colas. Cuando comienza una búsqueda, todas las peticiones están incluidas en una de las colas, estando la otra vacía. Durante la búsqueda, todas las nuevas peticiones se incluyen en la otra cola. Por tanto, se difiere el servicio de las nuevas peticiones hasta que se han procesado todas las peticiones antiguas.

11.6. RAID

Como se estudió anteriormente, el ritmo de mejora en el rendimiento del almacenamiento secundario ha sido considerablemente menor que en el caso de los procesadores y la memoria principal. Esta diferencia ha hecho del sistema de almacenamiento en disco probablemente el centro principal de interés para mejorar el rendimiento general del computador.

Como ocurre con otras áreas vinculadas con el rendimiento del computador, los diseñadores del almacenamiento de disco saben bien que si un determinado componente ya no puede mejorarse más, para lograr mejoras adicionales en el rendimiento se van a tener que utilizar múltiples componentes en paralelo. En el caso del almacenamiento de disco, esto lleva al desarrollo de vectores de discos que funcionan de manera independiente y en paralelo. Usando múltiples discos, se pueden manejar en paralelo las peticiones de E/S independientes, siempre que los datos solicitados residan en distintos discos. Además, una única petición de E/S se puede ejecutar en paralelo si el bloque de datos que se pretende acceder está distribuido a lo largo de múltiples discos.

Cuando se usan múltiples discos, hay muchas maneras de organizar los datos y la redundancia que se puede añadir para mejorar la fiabilidad. Esto podría dificultar el desarrollo de esquemas de ba-

ses de datos que se puedan utilizar en diversas plataformas y sistemas operativos. Afortunadamente, la industria ha definido un esquema estándar para el diseño de bases de datos en múltiples discos, conocido como RAID (*Redundant Array of Independent Disks*, Vector redundante de discos independientes). El esquema RAID consta de siete niveles[2], del cero al seis. Estos niveles no implican una relación jerárquica, sino que designan a diversas arquitecturas de diseño que comparten tres características comunes:

1. RAID corresponde con un conjunto de unidades físicas de disco tratado por el sistema operativo como un único dispositivo lógico.

2. Los datos están distribuidos a lo largo de las unidades físicas de un vector.

3. La capacidad de redundancia del disco se utiliza para almacenar información de paridad, que garantiza que los datos se pueden recuperar en caso de que falle un disco.

Los detalles de las características enumeradas en segundo y tercer lugar difieren en los distintos niveles RAID. RAID 0 y RAID 1 no incluyen la tercera característica.

El término *RAID* fue originalmente acuñado en un artículo de un grupo de investigadores de la universidad de California en Berkeley [PATT88][3]. El artículo definió varias configuraciones y aplicaciones de los discos RAID e introdujo las definiciones de los niveles RAID que se usan todavía hoy en día. Esta nueva estrategia remplazó los dispositivos de disco de gran capacidad por múltiples dispositivos de menor capacidad distribuyendo los datos de manera que se permitan los accesos simultáneos a los datos de múltiples dispositivos. Con ello, se mejora el rendimiento de E/S y se facilita el crecimiento gradual en la capacidad.

La contribución original de la propuesta RAID es abordar efectivamente la necesidad de redundancia. Aunque permitir que múltiples cabezas y activadores trabajen simultáneamente logra una mayor tasa de E/S y de transferencia, el uso de múltiples dispositivos incrementa la probabilidad de fallo. Para compensar esta pérdida de fiabilidad, RAID utiliza información de paridad almacenada que posibilita la recuperación de los datos perdidos debido a un fallo de un disco.

A continuación, se examinará cada uno de los niveles RAID. La Tabla 11.4, de [MASS97], proporciona un pequeño resumen de los siete niveles. En la tabla, el rendimiento de E/S se muestra tanto en términos de capacidad de transferencia de datos (es decir, posibilidad de mover datos) como de tasa de peticiones de E/S (es decir, capacidad de satisfacer peticiones de E/S), ya que estos niveles RAID presentan diferencias relativas inherentes con respecto a estos dos parámetros. El punto fuerte de cada nivel RAID está resaltado en color. La Figura 11.8 es un ejemplo que muestra el uso de los siete esquemas RAID para proporcionar una capacidad de datos equivalente a la de cuatro discos sin redundancia. La figura resalta la distribución de los datos de usuario y de redundancia e indica los requisitos de almacenamiento relativos de los distintos niveles. Se hará referencia a esta figura a lo largo del siguiente estudio.

---

[2] Algunos investigadores y algunas compañías han definido niveles adicionales. Sin embargo, los siete niveles definidos en esta sección son los conocidos universalmente.

[3] En ese artículo las siglas RAID significaban vector redundante de discos económicos (*Redundant Array of Inexpensive Disks*). El término *económico* se utilizaba para contrastar entre los discos pequeños relativamente baratos usados en el vector RAID y la alternativa consistente en un único disco caro y grande (SLED; *Single Large Expensive Disk*). La opción SLED es simplemente una cosa del pasado, puesto que actualmente se usa una tecnología de disco similar tanto para las configuraciones RAID como para las que no lo son. Por consiguiente, la industria ha adoptado el término *independiente* para enfatizar el hecho de que el vector RAID significa mejoras de rendimiento y fiabilidad significativas.

Tabla 11.4. Niveles RAID.

Categoría	Nivel	Descripción	Discos implicados	Disponibilidad de datos	Capacidad de transferencia para datos de E/S grandes	Tasa para peticiones de E/S pequeñas
En bandas	0	Sin redundancia	*N*	Inferior a un único disco	Muy alta	Muy alta tanto para lecturas como para escrituras
Espejo	1	Discos duplicados	2*N*, 3*N*, etc.	Mayor que RAID 2, 3, 4 o 5; menor que RAID 6	Mayor que un único disco para lecturas; similar a un único disco para escrituras	Hasta el doble de un único disco para lecturas; similar a un único disco para escrituras
Acceso paralelo	2	Redundancia mediante Código Hamming	*N* + *m*	Mucho mayor que un único disco; mayor que RAID 3, 4 o 5	La mayor de todas las alternativas mostradas	Aproximadamente el doble de un único disco
	3	Paridad intercalada a nivel de bit	*N* + 1	Mucho mayor que un único disco; comparable a RAID 2, 4 ó 5	La mayor de todas las alternativas mostradas	Aproximadamente el doble de un único disco
Acceso independiente	4	Paridad intercalada a nivel de bloque	*N* + 1	Mucho mayor que un único disco; comparable a RAID 2, 3 o 5	Similar a RAID 0 para lecturas; significativamente inferior a un único disco para escrituras	Similar a RAID 0 para lecturas; significativamente inferior a un único disco para escrituras
	5	Paridad distribuida e intercalada a nivel de bloque	*N* + 1	Mucho mayor que un único disco; comparable a RAID 2, 3 o 4	Similar a RAID 0 para lecturas; inferior a un único disco para escrituras	Similar a RAID 0 para lecturas; generalmente inferior a un único disco para escrituras
	6	Paridad dual distribuida e intercalada a nivel de bloque	*N* + 2	La mayor de todas las alternativas mostradas	Similar a RAID 0 para lecturas; inferior a RAID 5 para escrituras	Similar a RAID 0 para lecturas; significativamente inferior a RAID 5 para escrituras

RAID DE NIVEL 0

RAID de nivel 0 no es un verdadero miembro de la familia RAID, puesto que no incluye redundancia para mejorar la fiabilidad. Sin embargo, hay algunas aplicaciones, tales como algunas que se usan en los supercomputadores, en los que el rendimiento y la capacidad son primordiales, siendo más importante el bajo coste que la fiabilidad.

En RAID 0, los datos de los usuarios y del sistema están distribuidos a lo largo de todos los discos del vector. Esto tiene una importante ventaja sobre el uso de un único disco grande: si están pendientes dos peticiones de E/S diferentes que solicitan dos bloques de datos distintos, hay una gran probabilidad de que los bloques pedidos estén en diferentes discos. Por tanto, se pueden llevar a cabo las dos peticiones en paralelo, reduciendo el tiempo de espera en la cola de E/S.

Como ocurre con todos los otros niveles, RAID 0 hace algo más que simplemente distribuir los datos a lo largo de un vector de discos: Los datos están distribuidos *en bandas* (en inglés, *strips*) a lo largo de los discos disponibles. Esto se comprenderá mejor considerando la Figura 11.8. A todos los efectos, es como si los datos de los usuarios y del sistema estuvieran todos almacenados en un único disco lógico. El disco lógico está dividido en bandas; estas bandas pueden ser bloques físicos, sectores o alguna otra unidad. Las bandas se asignan de forma rotatoria a discos físicos consecutivos en el vector RAID. A un conjunto de bandas lógicamente consecutivas tal que a cada

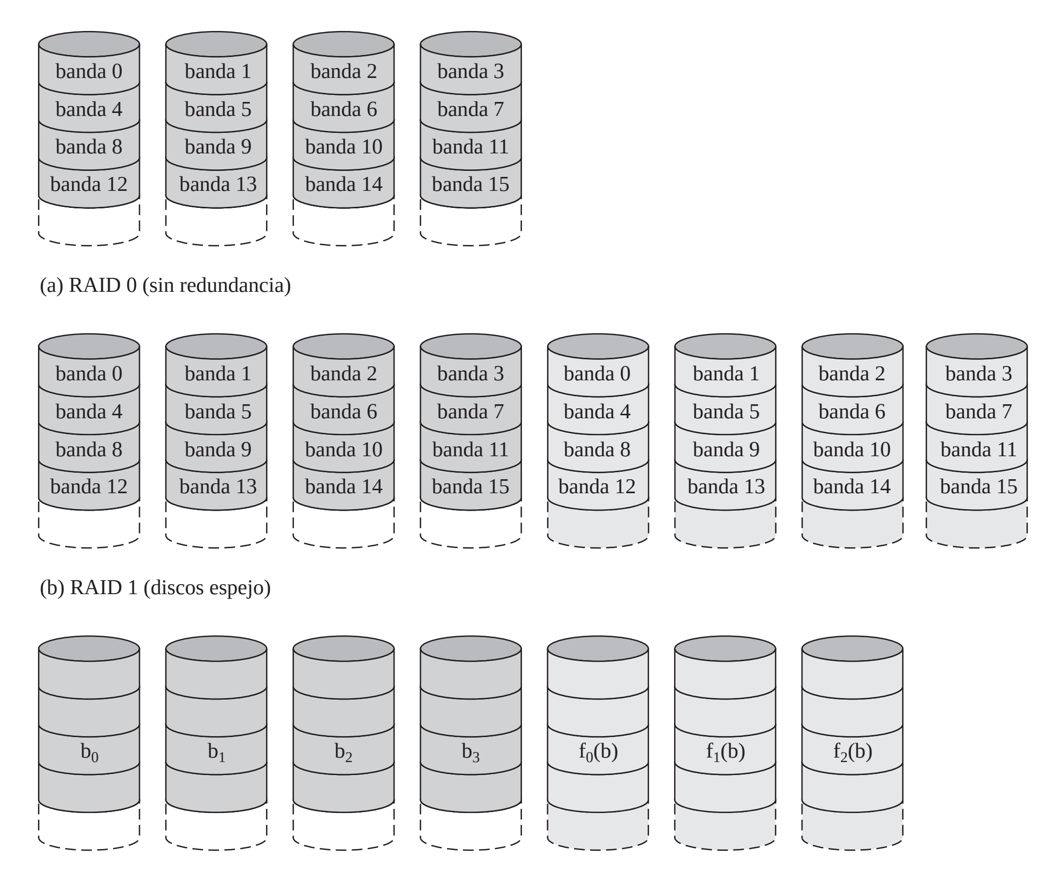


Figura 11.8. Niveles RAID (página 1 de 2).

miembro del vector de discos se le asigna exactamente una banda se le llama *lista* (en inglés, *stripe*). En un vector de *n* discos, las primeras *n* bandas lógicas se almacenan físicamente como la primera banda de cada uno de los *n* discos; las segundas *n* bandas lógicas se distribuyen constituyendo la segunda banda de cada disco; y así sucesivamente. La ventaja de esta distribución es que si una única petición de E/S consiste de múltiples bandas contiguas lógicamente, se pueden manejar en paralelo hasta *n* bandas de esta petición, reduciendo considerablemente el tiempo de transferencia de E/S.

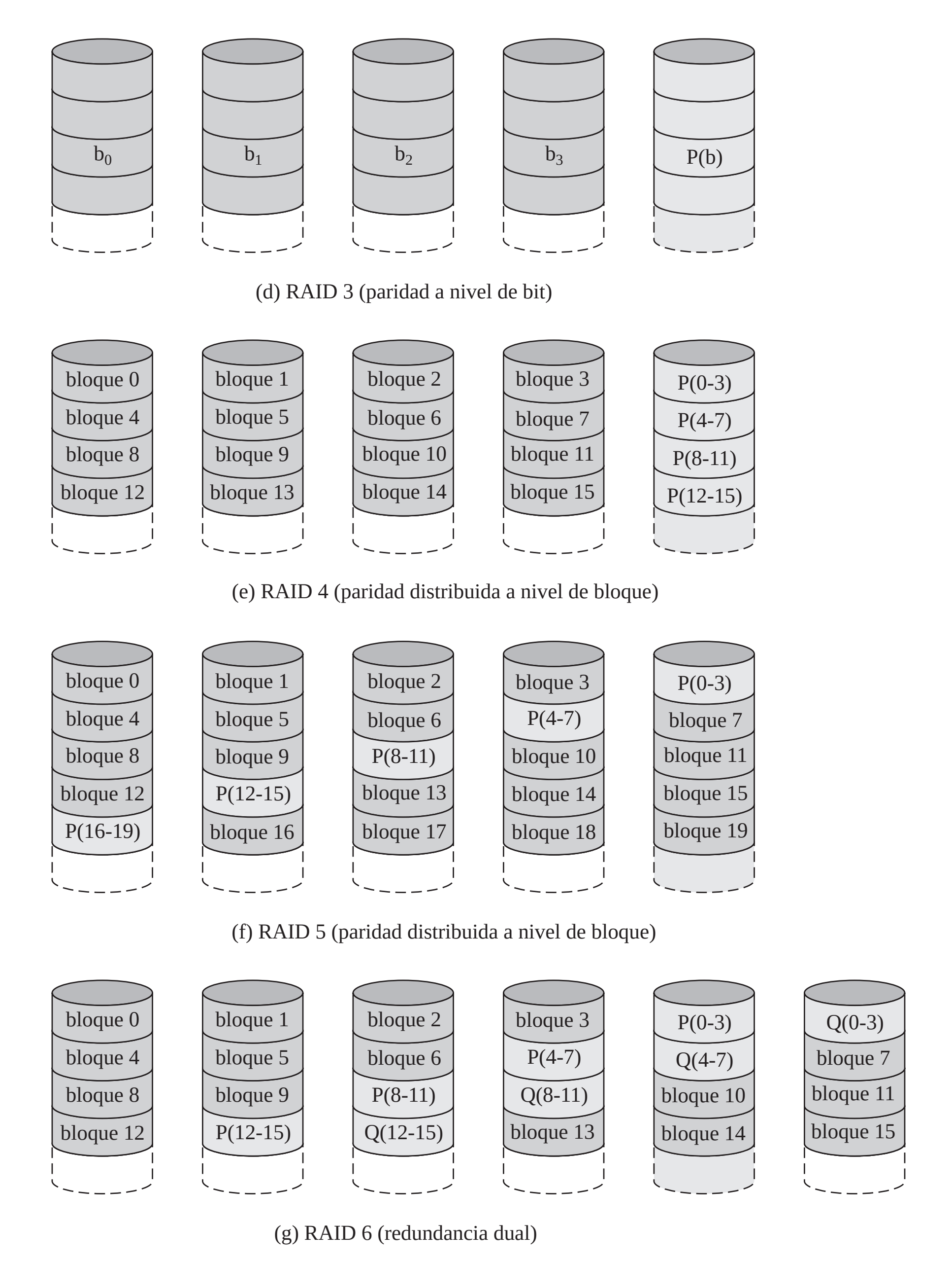


Figura 11.8. Niveles RAID (página 2 de 2).

RAID 0 para una elevada capacidad de transferencia de datos El rendimiento de cualquiera de los niveles RAID depende de forma crítica de los patrones de peticiones que haya en el sistema y de la distribución de los datos. Estos aspectos se pueden tratar más claramente en el nivel RAID 0, donde el impacto de la redundancia no interfiere en el análisis. En primer lugar, considérese el uso de RAID 0 para lograr una tasa de transferencia de datos elevada. Para que las aplicaciones experimenten esa alta tasa de transferencia, se deben cumplir dos requisitos. Primero, debe existir una elevada capacidad de transferencia a lo largo de todo el camino entre la memoria de la máquina y las unidades de disco individuales, incluyendo los buses internos del controlador, los buses de E/S del sistema, los adaptadores de E/S y los buses de memoria de la máquina.

El segundo requisito es que la aplicación debe hacer peticiones de E/S que accedan eficientemente al vector de discos. Este requisito se cumple si la mayoría de las peticiones corresponde con una gran cantidad de datos contiguos lógicamente, comparado con el tamaño de una banda. En este caso, una única petición de E/S involucra la transferencia de datos en paralelo desde múltiples discos, incrementando la tasa de transferencia efectiva comparada con una transferencia de un único disco.

RAID 0 para una elevada tasa de peticiones de E/S En un entorno orientado a transacciones, el usuario está normalmente más preocupado por el tiempo de respuesta que por la velocidad de transferencia. En el caso de una petición de E/S individual de una cantidad pequeña de datos, el tiempo de E/S está dominado por el movimiento de las cabezas del disco (tiempo de búsqueda) y el movimiento del disco (latencia rotacional).

En un entorno de transacciones, habrá cientos de peticiones de E/S por segundo. Un vector de discos puede proporcionar tasas elevadas de ejecución de E/S repartiendo la carga de E/S entre los múltiples discos. El reparto de carga efectivo sólo se alcanza si normalmente hay múltiples peticiones de E/S pendientes. Esto, a su vez, implica que hay múltiples aplicaciones independientes o una sola aplicación basada en transacciones que es capaz de realizar múltiples peticiones de E/S asíncronas. El rendimiento estará también influido por el tamaño de la banda. Si el tamaño de la banda es relativamente grande, de manera que una única petición de E/S sólo involucre un único acceso a disco, se pueden manejar en paralelo múltiples peticiones de E/S pendientes, reduciendo el tiempo de espera en la cola de cada petición.

RAID DE NIVEL 1

El nivel RAID 1 se diferencia de los niveles del 2 al 6 en el modo en que se logra la redundancia. En estos otros esquemas RAID, se utiliza algún tipo de cálculo de paridad para incluir la redundancia, mientras que en RAID 1, se consigue la redundancia mediante la simple estrategia de duplicar todos los datos. La Figura 11.8b muestra la distribución de datos utilizada, que es igual que la del esquema RAID 0. Sin embargo, en este caso cada banda lógica se asigna a dos discos físicos separados, de manera que cada disco en el vector tenga un disco duplicado que contenga los mismos datos. El esquema RAID 1 se puede implementar también sin la distribución de datos, aunque esto sea menos común.

Hay varios aspectos positivos en la organización RAID 1:

1. Una petición de lectura puede servirse de cualquiera de los dos discos que contienen los datos pedidos, aquél que implique un valor mínimo del tiempo de búsqueda más la latencia rotacional.
2. Una petición de escritura requiere actualizar ambas bandas, pero esto se puede hacer en paralelo. Por tanto, el rendimiento de la escritura lo establece el de la escritura más lenta (es decir, el que implica mayor valor del tiempo de búsqueda más la latencia rotacional). Sin embargo, no hay

una «penalización de escritura» con RAID 1. Los niveles RAID del 2 al 6 implican el uso de bits de paridad. Por tanto, cuando se actualiza una única banda, el software de gestión del vector debe primero calcular y actualizar los bits de paridad además de la banda real involucrada.

3. La recuperación de un fallo es sencilla. Cuando un dispositivo falla, todavía se puede acceder a los datos en el segundo dispositivo.

La principal desventaja de RAID 1 es el coste: requiere el doble de espacio de disco que el correspondiente al disco lógico proporcionado. Debido a esto, una configuración RAID 1 está probablemente limitada a dispositivos que almacenan software y datos del sistema u otros ficheros altamente críticos. En estos casos, RAID 1 proporciona una copia de respaldo en tiempo real de todos los datos de manera que, en el caso de un fallo de disco, todos los datos críticos están inmediatamente disponibles.

En un entorno de transacciones, RAID 1 puede alcanzar tasas elevadas de peticiones de E/S si la mayor parte de las peticiones son de lectura. En esta situación, el rendimiento de RAID 1 puede acercarse al doble del proporcionado por el esquema RAID 0. Sin embargo, si una parte sustancial de las peticiones de E/S son de escritura, puede que no haya ganancia de rendimiento con respecto a RAID 0. RAID 1 puede también proporcionar un mejor rendimiento que RAID 0 para aplicaciones en las que hay transferencias intensivas de datos con un elevado porcentaje de lecturas. La mejora se produce si la aplicación puede dividir cada petición de lectura de manera que participen ambos grupos de discos.

RAID DE NIVEL 2

Los niveles RAID 2 y 3 utilizan una técnica de acceso paralelo. En un vector de acceso paralelo, todos los miembros del disco participan en la ejecución de cada petición de E/S. Normalmente, los ejes de las distintas unidades se sincronizan de manera que en todo momento la cabeza de cada disco esté en la misma posición en todos los discos.

Como ocurre con los otros esquemas RAID, se utiliza una distribución de datos en bandas. En el caso de los esquemas RAID 2 y 3, las bandas son muy pequeñas, a menudo un único byte o palabra. Con RAID 2, se calcula un código de corrección de error con los bits correspondientes de cada disco de datos, almacenándose los bits del código resultante en las correspondientes posiciones de bit en los múltiples discos de paridad. Normalmente, se utiliza un Código Hamming, que es capaz de corregir errores en un único bit y detectar errores en dos.

Aunque RAID 2 requiere menos discos que RAID 1, sigue siendo bastante costoso. El número de discos redundantes es proporcional al logaritmo del número de discos de datos. En una única lectura, se acceden todos los discos de manera simultánea. El controlador del vector recibe los datos pedidos y el código de corrección de error asociado. Si hay un error en un único bit, el controlador puede detectar y corregir el error instantáneamente, con lo que el tiempo de acceso de lectura no se ralentiza. En una única escritura, la operación de escritura debe acceder a todos los discos de datos y de paridad para llevarse a cabo.

El esquema RAID 2 sólo sería una opción efectiva en un entorno en el que se produjeran muchos errores de disco. Dada la alta fiabilidad de las unidades de disco y de los discos propiamente dichos, el esquema RAID 2 es excesivo y no se implementa en la práctica.

RAID DE NIVEL 3

El esquema RAID 3 se organiza de una manera similar al usado en RAID 2. La diferencia estriba en que RAID 3 requiere sólo un disco redundante, con independencia del tamaño del vector de discos.

RAID 3 emplea acceso paralelo, teniendo los datos distribuidos en pequeñas bandas. En lugar de un código de corrección de errores, se calcula un bit de paridad simple para el conjunto de bits almacenados en la misma posición en todos los discos de datos.

Redundancia En el caso de que ocurra un fallo en un dispositivo, se accede al dispositivo de paridad y se reconstruyen los datos desde los dispositivos restantes. Una vez que se remplaza el dispositivo que falló, los datos perdidos pueden restaurarse en el nuevo dispositivo y la operación se reanuda.

La reconstrucción de datos es simple. Considere un vector con cinco unidades de tal manera que los dispositivos que van desde X0 a X3 contienen datos, mientras que la unidad X4 es el disco de paridad. La paridad para el bit *i*-ésimo se calcula de la siguiente manera:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

donde $\oplus$ es una función O-exclusivo.

Supóngase que falla el dispositivo X1. Si se suma $X4(i) \oplus X1(i)$ a ambos lados de la ecuación precedente, se obtiene:

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Por consiguiente, los contenidos de cada banda de datos en X1 pueden regenerarse a partir de los contenidos de las bandas correspondientes en los restantes discos del vector. Este principio es aplicable a los niveles RAID desde el 3 al 6.

Si se produce un fallo en un disco, todos los datos están todavía disponibles en lo que se denomina modo reducido. En este modo, para las lecturas, los datos perdidos se regeneran sobre la marcha utilizando el cálculo del O-exclusivo. Cuando se escriben datos en un vector RAID 3 en modo reducido, se debe mantener la coherencia de paridad para una posterior regeneración. El retorno a un modo de operación pleno requiere que se remplace el disco que falló y se regenere el contenido completo de dicho disco en el disco nuevo.

Rendimiento Dado que los datos están distribuidos en pequeñas bandas, el esquema RAID 3 puede alcanzar velocidades de transferencia de datos muy elevadas. Cualquier petición de E/S involucrará la transferencia en paralelo de datos de todos los discos de datos. Para transferencias grandes, la mejora en el rendimiento es especialmente notable. Por otro lado, sólo se puede ejecutar una petición de E/S en cada momento. Por tanto, en entorno de transacciones, se ve afectado el rendimiento.

RAID DE NIVEL 4

Los niveles RAID de 4 a 6 utilizan una técnica de acceso independiente. En un vector de acceso independiente, cada disco del vector opera independientemente, de manera que se pueden servir en paralelo peticiones de E/S independientes. Debido a esto, los vectores de acceso independiente son más adecuados para las aplicaciones que requieren tasas elevadas de peticiones de E/S y relativamente menos adecuados para aquéllas que necesitan tasas elevadas de transferencias de datos.

Como ocurre en los otros esquemas RAID, se utiliza una distribución de datos en bandas. En el caso de los esquemas RAID desde el 4 al 6, las bandas son relativamente grandes. En RAID 4, se calcula bit a bit una banda de paridad a partir de las bandas correspondientes de cada disco de datos, almacenándose los bits de paridad resultantes en la banda correspondiente del disco de paridad.

El esquema RAID 4 implica una penalización a las escrituras cuando son de tamaño pequeño. Cada vez que se produce una escritura, el software de gestión del vector, además de modificar los da-

tos de usuario involucrados, debe actualizar los datos de paridad correspondientes. Considere un vector con cinco unidades de tal manera que los dispositivos que van desde X0 a X3 contienen datos, mientras que la unidad X4 es el disco de paridad. Supóngase que se realiza una escritura que sólo involucra una banda en el disco X1. Inicialmente, para cada bit *i* se cumplirá la siguiente relación:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \tag{11.1}$$

A continuación, se muestra el resultado después de la actualización, donde se ha marcado con una comilla los bits potencialmente modificados.

$$\begin{aligned} X4(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

El conjunto precedente de ecuaciones se deriva como se explica a continuación. La primera línea muestra que un cambio en X1 afectará también al disco de paridad X4. En la segunda línea, se suman los términos [$\oplus$ X1(i) $\oplus$ X1(i)]. Dado que el O-exclusivo de cualquier valor consigo mismo es 0, esto no afecta a la ecuación. Sin embargo, es un paso intermedio conveniente para llegar a la tercera línea, donde simplemente se cambia el orden de los términos. Finalmente, se utiliza la ecuación (11.1) para remplazar los cuatro primeros términos por X4(i).

Para calcular la nueva paridad, el software de gestión del vector debe leer las bandas de datos y paridad antiguas. A continuación, puede actualizar estas dos bandas con los nuevos datos y la nueva paridad calculada. Por consiguiente, cada escritura en una banda requiere dos lecturas y dos escrituras.

En el caso de una escritura de E/S de mayor tamaño que involucre bandas en todas las unidades de disco, la paridad se calcula fácilmente utilizando sólo los nuevos bits de datos. Por tanto, el dispositivo de paridad se puede actualizar en paralelo a los dispositivos de datos sin requerir lecturas o escrituras adicionales.

En cualquier caso, cada operación de escritura debe involucrar el disco de paridad, que, por tanto, puede llegar a ser un cuello de botella.

RAID DE NIVEL 5

El esquema RAID 5 se organiza de manera similar al RAID 4. La diferencia estriba en que el esquema RAID 5 distribuye las bandas de paridad a través de todos los discos. La asignación habitual usa un esquema rotatorio, como se muestra en la Figura 11.8f. Para un vector de *n* discos, la banda de paridad está en un disco diferente para las *n* primeras listas, y, a continuación, se repite este patrón de asignación.

La distribución de las bandas de paridad a través de todos los dispositivos evita el potencial cuello de botella de E/S debido a la existencia de un único disco de paridad que aparece en el esquema RAID 4.

RAID DE NIVEL 6

El esquema RAID 6 fue propuesto en un artículo posterior por los investigadores de Berkeley [KATZ89]. En el esquema RAID 6, se realizan dos cálculos de paridad diferentes, almacenándose en

bloques separados de distintos discos. Por tanto, un vector RAID 6, cuyos datos de usuario requieran *N* discos, necesitará *N* + 2 discos.

La Figura 11.8g muestra este esquema. P y Q son dos algoritmos de comprobación de datos diferentes. Uno de los dos es el cálculo O-exclusivo usado en los esquemas RAID 4 y 5. Sin embargo, el otro es un algoritmo de comprobación de datos independiente. Esto permite regenerar los datos incluso si fallan dos discos que contienen datos de usuario.

La ventaja del esquema RAID 6 es que proporciona una extremadamente alta disponibilidad de datos. Tendrían que fallar tres discos dentro del intervalo correspondiente al tiempo medio de reparación (*Mean Time To Repair*, MTTR) para causar una pérdida de datos. Por otro lado, el esquema RAID 6 incurre en una penalización de escritura sustancial, debido a que cada escritura afecta a dos bloques de paridad.

11.7. CACHE DE DISCO

En la Sección 1.6 y en el Apéndice 1A, se resumen los fundamentos de las memorias cache. El término *memoria cache* se utiliza normalmente aplicado a una memoria que es más pequeña y más rápida que la memoria principal y que se interpone entre la memoria principal y el procesador. Dicha memoria cache reduce el tiempo medio de acceso a memoria explotando el principio de la proximidad.

El mismo principio se puede aplicar a la memoria del disco. Específicamente, una cache de disco es un *buffer* en memoria principal para almacenar sectores del disco. La cache contiene una copia de algunos de los sectores del disco. Cuando se hace una petición de E/S solicitando un determinado sector, se comprueba si el sector está en la cache del disco. En caso afirmativo, se sirve la petición desde la cache. Debido al fenómeno de la proximidad de referencias, cuando se lee un bloque de datos en la cache para satisfacer una única petición de E/S, es probable que haya referencias a ese mismo bloque en el futuro.

CONSIDERACIONES DE DISEÑO

Hay varios aspectos de diseño que son de interés. En primer lugar, cuando se satisface una petición de E/S de la cache de disco, se deben entregar los datos de la cache al proceso solicitante. Esta operación se puede hacer o bien copiando el bloque de datos almacenado en la memoria principal asignada a la cache de disco hasta la memoria asignada al proceso de usuario, o bien utilizando simplemente la técnica de la memoria compartida pasando un puntero al bloque correspondiente de la cache de disco. Esta última estrategia ahorra el tiempo de la transferencia de memoria a memoria y también permite el acceso compartido por parte de otros procesos utilizando el modelo de lectores/escritores descrito en el Capítulo 5.

Un segundo aspecto de diseño está relacionado con la estrategia de remplazo. Cuando se trae un nuevo sector a la cache de disco, se debe remplazar uno de los bloques existentes. Se trata de un problema idéntico al presentado en el Capítulo 8, donde se requería un algoritmo de remplazo de páginas. Se han probado diversos algoritmos. El algoritmo más frecuentemente utilizado es el del menos recientemente usado (*Least Recently Used*, LRU): se remplaza el bloque que ha estado en la cache más tiempo sin ser accedido. Lógicamente, la cache consiste de una pila de bloques, estando el bloque más recientemente accedido en lo más alto de la pila. Cuando se accede a un bloque en la cache, se mueve de su posición actual en la pila hasta la cima de la misma. Cuando se trae un bloque desde la memoria secundaria, se elimina el bloque que estaba al final de la pila, situándose el bloque entrante en la cima de la pila. Naturalmente, no es necesario mover realmente estos bloques dentro de la memoria principal, puesto que se puede asociar una pila de punteros a la cache.

Otra posibilidad es el algoritmo del **menos frecuentemente usado (*Least Frequently Used*, LFU).** Se remplaza el bloque del conjunto que ha experimentado la menor cantidad de referencias. El algoritmo LFU puede implementarse asociando un contador con cada bloque. Cuando se trae un bloque, se le asigna un contador con un valor igual a 1, incrementando el contador en 1 por cada referencia al bloque. Cuando se requiere un remplazo, se selecciona el bloque con un contador más pequeño. Intuitivamente, podría parece que LFU es más apropiado que LRU porque utiliza información más pertinente de cada bloque en el proceso de selección.

Un algoritmo LFU sencillo conlleva el siguiente problema. Puede ocurrir que algunos bloques se accedan en términos globales de forma relativamente infrecuente, pero cuando se hace referencia a ellos, se producen referencias repetidas durante cortos intervalos de tiempo debido a la proximidad, de manera que se genera un contador de referencias elevado. Cuando se acaba el intervalo, el valor del contador de referencias puede llevar a conclusiones erróneas, no reflejando el grado de probabilidad de que el bloque se acceda de nuevo en un breve plazo de tiempo. Por tanto, el efecto de la proximidad puede realmente causar que el algoritmo LFU tome decisiones inadecuadas.

Para resolver esta deficiencia del algoritmo LFU, en [ROBI90] se propone una técnica denominada remplazo basado en la frecuencia. Por motivos pedagógicos, considere una versión simplificada, mostrada en la Figura 11.9a. Los bloques se organizan lógicamente en una pila, como en el algoritmo LRU. Una cierta porción de la parte superior de la pila se considera separada como una sección nueva. Cuando hay un acierto en la cache, el bloque accedido se mueve a la parte superior de la pila. Si el bloque ya estaba en la sección nueva, su contador de referencias no se incrementa; en caso contrario, se incrementa en 1. Si la sección nueva es suficientemente grande, se consigue que permanezcan inalterados los contadores de referencias de los bloques que se acceden repetidamente dentro de un breve intervalo de tiempo. En caso de fallo, se escoge para remplazar el bloque con el contador de referencias más bajo que no esté en la sección nueva, seleccionando el bloque usado menos recientemente en caso de empate.

Los autores explican que esta estrategia sólo logra una ligera mejora con respecto al algoritmo LRU. El problema es el siguiente:

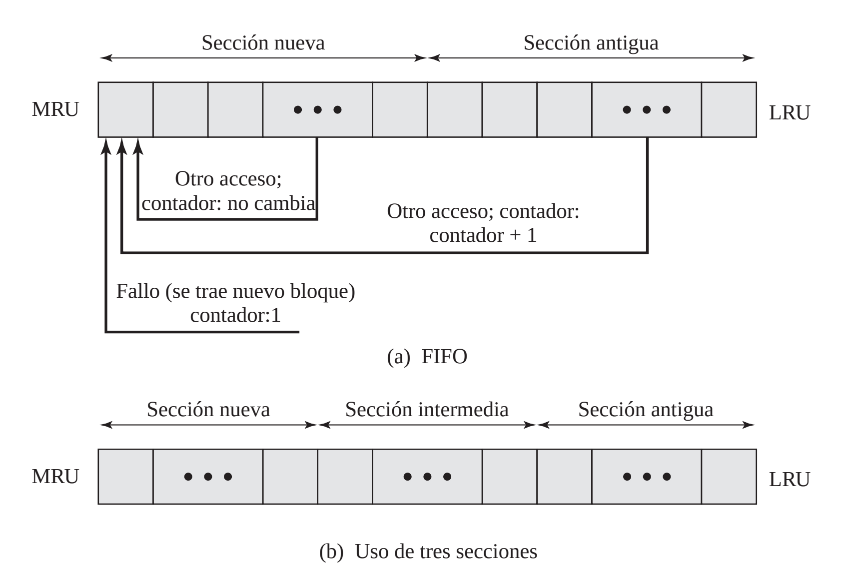


Figura 11.9. Remplazo basado en la frecuencia.

1. En un fallo de cache, se incluirá un nuevo bloque en la sección nueva, con un contador igual a 1.
2. El contador continúa con un valor igual a 1 mientras el bloque permanezca en la sección nueva.
3. Pasado un cierto tiempo, el bloque sale de la sección nueva, con su contador todavía igual a 1.
4. Si no se vuelve a acceder al bloque con cierta prontitud, es muy probable que se remplace porque tiene necesariamente el menor contador de referencias entre todos los bloques que no están en la sección nueva. En otras palabras, parece que no hay un intervalo suficientemente largo para que los bloques que abandonan la sección nueva puedan aumentar el valor de su contador de referencias incluso aunque se accedan con una cierta frecuencia.

Una mejora adicional afronta este problema: dividir la pila en tres secciones: nueva, intermedia y antigua. Como antes, los contadores de referencias no se incrementan en el caso de bloques incluidos en la sección nueva. Sin embargo, sólo se seleccionan para remplazo los bloques en la sección antigua. Asumiendo una sección intermedia grande, este esquema proporciona a los bloques que se accedan con una cierta frecuencia una oportunidad para aumentar el valor de sus contadores de referencias antes de llegar a ser candidatos al remplazo. Las simulaciones realizadas por los autores indican que esta política mejorada es significativamente mejor que LRU o LFU.

Con independencia de cada estrategia de remplazo específica, el remplazo puede realizarse bajo demanda o planificado por anticipado. En el primer caso, sólo se remplaza un sector cuando se necesita un hueco. En el último caso, se generan simultáneamente varios huecos. El motivo de esta última estrategia está relacionado con la necesidad de volver a escribir los sectores en el disco. Si se trae a la cache un sector y sólo se lee, cuando se remplace no será necesario escribirlo de nuevo en el disco. Sin embargo, si se actualiza el sector, es necesario volverlo a escribir antes de remplazarlo. En este último caso, es razonable agrupar las escrituras y ordenarlas para minimizar el tiempo de búsqueda.

CONSIDERACIONES DE RENDIMIENTO

En este caso se aplican las mismas consideraciones de rendimiento analizadas en el Apéndice 1A. El aspecto del rendimiento de la cache se reduce en sí mismo a una cuestión de si se puede alcanzar una determinada tasa de fallos. Esto dependerá del comportamiento de las referencias al disco con respecto a la proximidad y del algoritmo de remplazo, así como de otros factores de diseño. Sin embargo, la tasa de fallos principalmente está en función del tamaño de la cache del disco. La Figura 11.10 resume los resultados de varios estudios usando LRU, uno para un sistema UNIX ejecutado en un VAX [OUST85] y otro para sistemas operativos de *mainframes* de IBM [SMIT85]. La Figura 11.11 muestra los resultados de simulaciones del algoritmo de remplazo basado en la frecuencia. Una comparación de las dos figuras resalta uno de los riesgos de este tipo de evaluación del rendimiento. Las figuras parecen mostrar que el algoritmo LRU supera al algoritmo de remplazo basado en la frecuencia. Sin embargo, cuando se comparan patrones de referencias idénticos utilizando la misma estructura de cache, el algoritmo de remplazo basado en la frecuencia es superior. Por tanto, la secuencia exacta de patrones de referencia, así como los aspectos de diseño relacionados como el tamaño de bloque, tendrá una profunda influencia en el rendimiento alcanzado.

11.8. E/S DE UNIX SVR4

En UNIX, cada dispositivo de E/S está asociado con un fichero especial, que lo gestiona el sistema de ficheros y se lee y escribe de la misma manera que los ficheros de datos de usuario. Esto proporciona una interfaz bien definida y uniforme para los usuarios y los procesos. Para leer o escribir

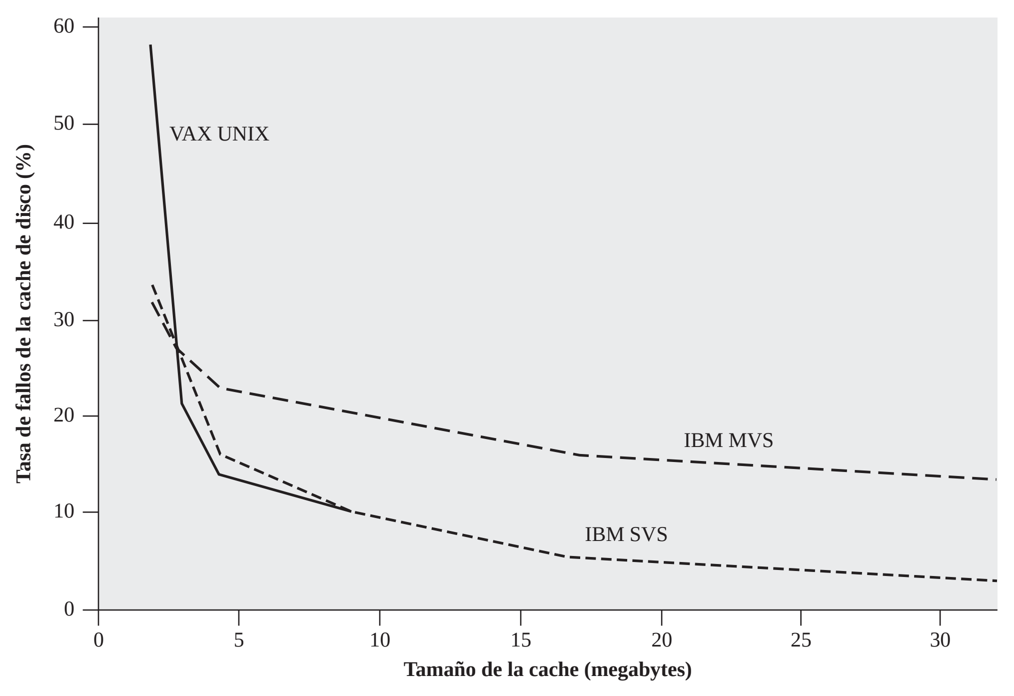


Figura 11.10. Algunos resultados del rendimiento de una cache de disco usando LRU.

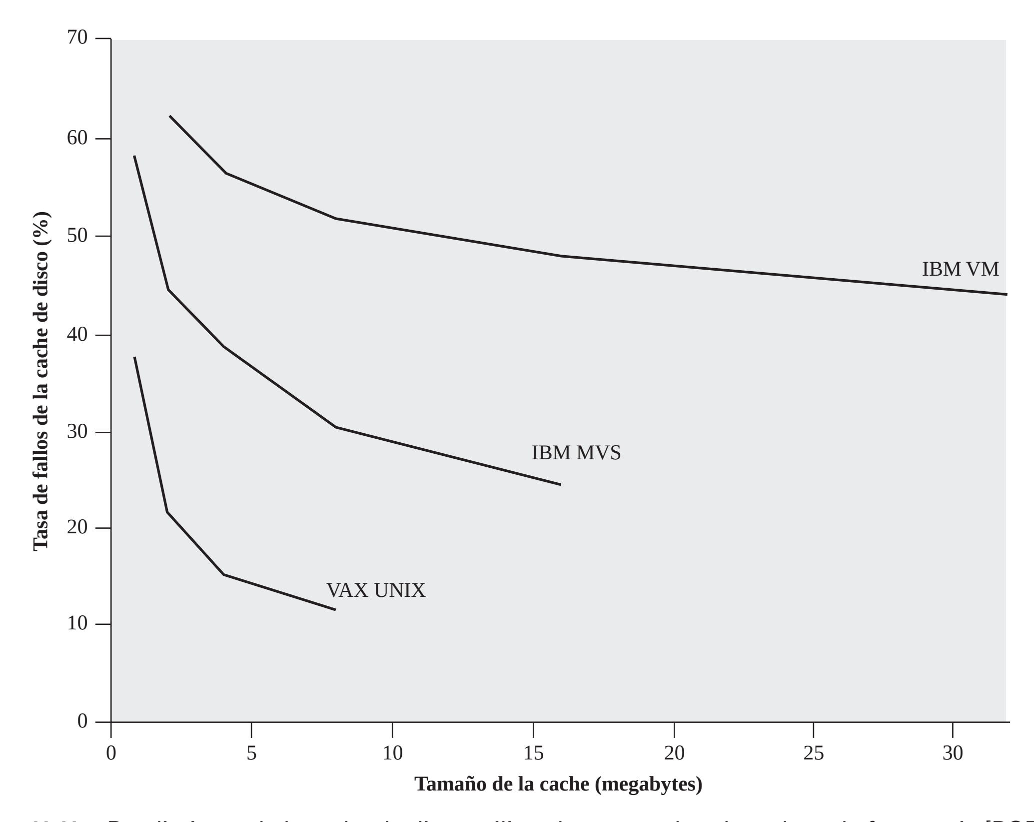


Figura 11.11. Rendimiento de la cache de disco utilizando un remplazo basado en la frecuencia [ROBI90].

de un dispositivo, se realizan peticiones de lectura o escritura en el fichero especial asociado con el dispositivo.

La Figura 11.12 muestra la estructura lógica del sistema de E/S. El subsistema de ficheros gestiona los ficheros en los dispositivos de almacenamiento secundario. Además, sirve como interfaz a los dispositivos para los procesos, debido a que se tratan como ficheros.

Hay dos tipos de E/S en UNIX: con *buffer* y sin *buffer*. La E/S con *buffer* pasa a través de los *buffers* del sistema, mientras que la E/S sin *buffer* normalmente involucra al sistema de DMA, de manera que la transferencia tiene lugar directamente entre el módulo de E/S y el área de E/S del proceso. En la E/S con *buffer*, se utilizan dos tipos de *buffers*: caches de *buffers* del sistema y colas de caracteres.

CACHE DE *BUFFERS*

La cache de *buffers* de UNIX es esencialmente una cache de disco. Las operaciones de E/S sobre el disco se manejan a través de la cache de *buffers*. La transferencia de datos entre la cache de *buffers* y el espacio del proceso de usuario siempre se realiza utilizando DMA. Dado que tanto la cache de *buffers* como el área de E/S del proceso están en la memoria principal, se utiliza en este caso el sistema de DMA para realizar una copia de memoria a memoria. Esta operación no consumirá ningún ciclo del procesador, pero consume ciclos del bus.

Para gestionar la cache de *buffers* se usan tres listas:

- **Lista de libres.** Lista de todos los huecos de la cache (en UNIX a un hueco se le denomina *buffer*, cada hueco almacena un sector del disco) que están disponibles para su asignación.
- **Lista de dispositivos.** Lista de todos los *buffers* que están actualmente asociados con cada disco.
- **Cola de E/S del manejador.** Lista de los *buffers* sobre los que se está realmente realizando E/S o esperando por la misma para un determinado dispositivo.

Todos los *buffers* deberían estar en la lista de libres o en la cola de E/S del manejador. Una vez que se asocia un *buffer* a un dispositivo, permanece asociado al mismo incluso aunque esté en la lista de libres, hasta que se reutilice realmente y pase a estar vinculado con otro dispositivo. Estas listas se gestionan usando punteros asociados a cada *buffer* en vez de utilizar listas físicamente independientes.

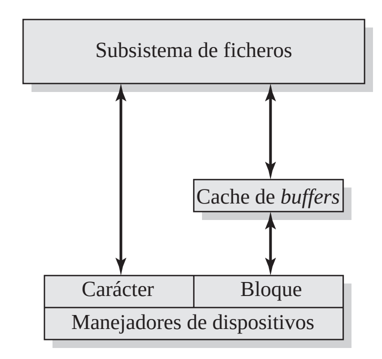


Figura 11.12. Estructura de la E/S de UNIX.

Cuando se hace una referencia a un número de bloque físico de un determinado dispositivo, el sistema operativo primero comprueba si el bloque está en la cache de *buffers*. Para minimizar el tiempo de búsqueda, la lista de dispositivos se organiza como una tabla *hash*, utilizando una técnica similar al desbordamiento con encadenamiento estudiado en el Apéndice 8A (Figura 8.27b). La Figura 11.13 muestra la organización general de la cache de *buffers*. Hay una tabla *hash* de longitud fija que contiene punteros a la cache de *buffers*. Cada referencia a un (dispositivo#, bloque#) se corresponde con una determinada entrada en la tabla *hash*. El puntero de dicha entrada apunta al primer *buffer* de la cadena. Un puntero *hash* incluido en cada *buffer* señala al siguiente *buffer* en la cadena correspondiente a esa entrada de la tabla *hash*. Por tanto, se cumple que para todas las referencias (dispositivo#, bloque#) que se corresponden con la misma entrada de la tabla *hash*, si el bloque correspondiente está en la cache de *buffers*, ese *buffer* estará en la cadena vinculada a esa entrada de la tabla *hash*. Por consiguiente, la longitud de la búsqueda en la cache de *buffers* se reduce por un factor de orden *N*, siendo *N* la longitud de la tabla *hash*.

Para el remplazo de un bloque, se utiliza el algoritmo del menos recientemente usado. Una vez que a un *buffer* se le ha asignado un bloque de disco, no puede utilizarse para otro bloque hasta que se hayan usado más recientemente los restantes *buffers*. La lista de libres mantiene el orden requerido por el algoritmo.

COLA DE CARACTERES

Los dispositivos orientados a bloques, como el disco y la cinta, se pueden gestionar eficientemente usando la cache de *buffers*. Sin embargo, los dispositivos orientados a caracteres, como los terminales y las impresoras, requieren otro tipo de *buffers*: las colas de caracteres. Una cola de caracteres, en

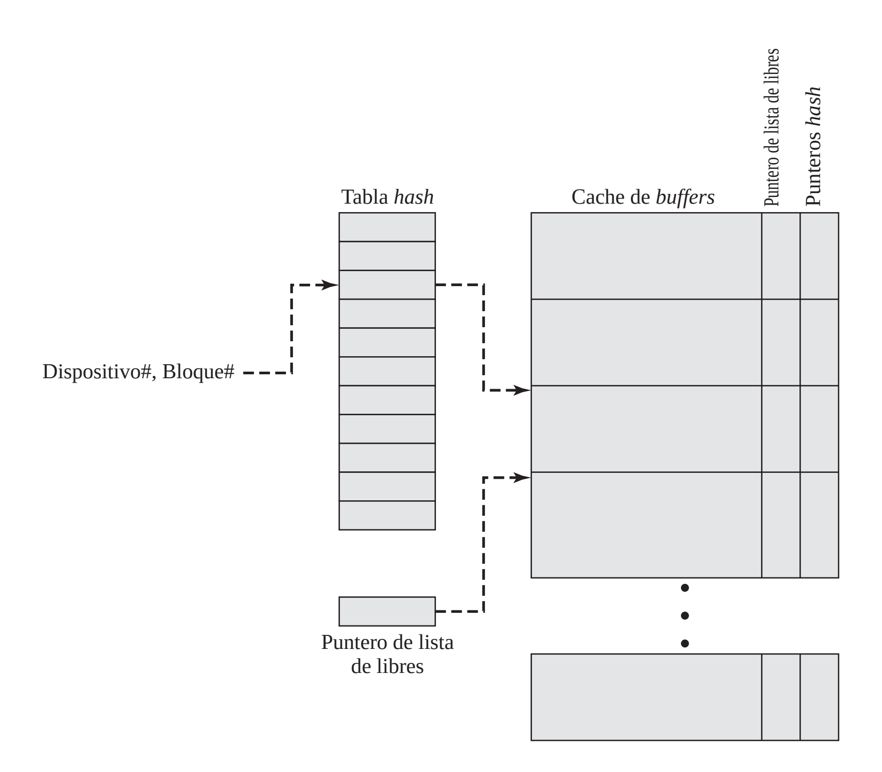


Figura 11.13. Organización de la cache de *buffers* en UNIX.

ciertos casos, es escrita por el dispositivo de E/S y leída por el proceso, mientras que en otras ocasiones es el proceso el que escribe y el dispositivo el que lee. En ambos casos, se utiliza el modelo productor/consumidor presentado en el Capítulo 5. Por tanto, las colas de caracteres sólo se pueden leer una vez; cuando se lee un carácter, de hecho, se destruye. Esto contrasta con la cache de *buffers*, que puede leerse múltiples veces y, por tanto, corresponde con el modelo lectores/escritores (también estudiado en el Capítulo 5).

E/S SIN *BUFFER*

La E/S sin *buffer*, que consiste simplemente en una operación de DMA entre el dispositivo y el espacio del proceso, es siempre el método más rápido de realizar E/S por parte de un proceso. Un proceso que está realizando E/S sin *buffer* queda residente en la memoria principal, no pudiendo resultar expulsado. Esto reduce la oportunidad de usar el intercambio al fijar como residente parte de la memoria principal, lo que conlleva una disminución del rendimiento global del sistema. Asimismo, el dispositivo de E/S queda asociado al proceso durante la duración de la transferencia, no estando disponible mientras tanto para otros procesos.

DISPOSITIVOS DE UNIX

Entre las categorías de dispositivos reconocidos por UNIX se encuentran las siguientes:

- Dispositivos de disco.
- Dispositivos de cinta.
- Terminales.
- Líneas de comunicación.
- Impresoras.

La Tabla 11.5 muestra los tipos de E/S apropiados para cada tipo de dispositivo. Los dispositivos de disco, que se utilizan abundantemente en UNIX, son orientados a bloques y, en principio, suelen tener un rendimiento razonablemente bueno. Por tanto, la E/S de estos dispositivos suele ser sin *buffer* o usando la cache de *buffers*. Los dispositivos de cinta son funcionalmente similares a los dispositivos de disco utilizando unos esquemas de E/S similares.

Tabla 11.5. E/S de dispositivos en UNIX.

	E/S sin *buffer*	Cache de *buffers*	Cola de caracteres
Unidad de disco	X	X	
Unidad de cinta	X	X	
Terminales			X
Líneas de comunicación			X
Impresoras	X		X

Dado que los terminales requieren un intercambio relativamente lento de caracteres, la E/S de terminal utiliza normalmente la cola de caracteres. De manera similar, las líneas de comunicación re-

quieren el procesamiento en serie de bytes de datos de entrada o salida, por lo que se manejan mejor con colas de caracteres. Por último, el tipo de E/S utilizado por una impresora dependerá generalmente de su velocidad. Las impresoras lentas utilizan normalmente colas de caracteres, mientras que una rápida puede emplear E/S sin *buffer*. En una impresora rápida puede utilizarse una cache de *buffers*. Sin embargo, dado que los datos que van a la impresora nunca se reutilizan, la sobrecarga de la cache de *buffers* es innecesaria.

11.9. E/S DE LINUX

En términos generales, el sistema de E/S del núcleo de Linux es muy similar al de otras implementaciones de UNIX, como es el caso de SVR4. El núcleo de Linux asocia un fichero especial con cada manejador de dispositivo de E/S, distinguiéndose entre dispositivos de bloques, de caracteres y de red. En esta sección, se estudiarán varias características del sistema de E/S de Linux.

PLANIFICACIÓN DE DISCO

El planificador de disco por defecto en Linux 2.4 se le conoce con el nombre de ascensor de Linus, que es una variación del algoritmo LOOK estudiado en la Sección 11.3. En Linux 2.6, además del algoritmo del ascensor, se han incluido dos algoritmos adicionales: el planificador de E/S basado en plazos y el planificador de E/S previsor [LOVE04b]. A continuación, se estudiará cada uno de ellos.

EL PLANIFICADOR DEL ASCENSOR

El planificador del ascensor mantiene una única cola con las peticiones de lectura y escritura en el disco, realizando operaciones de ordenamiento y agrupamiento sobre la cola. En términos generales, el planificador del ascensor mantiene la lista de peticiones ordenadas por el número de bloque. De esta manera, cuando se manejan las peticiones de disco, el dispositivo se mueve en una única dirección, satisfaciendo cada petición según la encuentra. Esta estrategia general se mejora de la siguiente manera. Cuando se añade una nueva petición a la cola, se consideran en este orden las siguientes cuatro operaciones:

1. Si existe una petición pendiente en la cola de tal manera que la nueva petición corresponde con el mismo sector del disco o uno inmediatamente adyacente al requerido por dicha petición previa, la petición existente y la nueva se mezclan en una sola.
2. Si hay una petición en la cola que es suficientemente antigua, la nueva petición se inserta al final de la cola.
3. Si hay una posición adecuada, la nueva petición se sitúa en el orden correspondiente.
4. Si no hay una posición adecuada, la nueva petición se sitúa al final de la cola.

PLANIFICADOR BASADO EN PLAZOS

La segunda operación de la lista precedente intenta evitar la inanición de una petición, pero no es muy efectiva [LOVE04a]. No intenta servir peticiones en un plazo de tiempo determinado, sino que simplemente deja de insertar las peticiones en orden después de un plazo conveniente. En el esquema del ascensor se manifiestan dos problemas. El primer problema es que se puede retrasar una petición

de un bloque distante durante un tiempo considerable debido a que la cola se actualiza dinámicamente. Por ejemplo, considere el siguiente flujo de peticiones de los bloques de disco: 20, 30, 700 y 25. El planificador del ascensor modifica el orden de estas peticiones de manera que se sitúan en la cola en el orden 20, 25, 30 y 700, estando la petición del bloque 20 en la cabeza de la cola. Si llega una secuencia continua de peticiones que corresponden con bloques de baja numeración, la petición del bloque 700 se retrasa indefinidamente.

Un problema incluso más serio es el de la distinción entre peticiones de lectura y de escritura. Normalmente, una petición de escritura se realiza asíncronamente. Es decir, una vez que un proceso solicita una petición de escritura, no necesita esperar hasta que realmente se lleve a cabo la petición. Cuando una aplicación solicita una escritura, el núcleo copia los datos en un *buffer* apropiado, que se escribirá cuando se considere oportuno. Una vez que se copian los datos en el *buffer* del núcleo, la aplicación puede continuar. Sin embargo, en muchas operaciones de lectura, el proceso, antes de continuar, debe esperar hasta que se entreguen los datos pedidos a la aplicación. Por tanto, un flujo de peticiones de escritura (por ejemplo, para escribir en el disco un fichero grande) puede bloquear una petición de lectura durante un tiempo considerable y, con ello, bloquear también al proceso.

Para resolver estos problemas, se utiliza el planificador de E/S basado en plazos que usa tres colas (Figura 11.14). Cada nueva petición se incluye en la cola ordenada del ascensor, como en el algoritmo previo. Asimismo, esa misma petición se sitúa al final de una cola FIFO de lectura en el caso de una petición de lectura o de una cola FIFO de escritura si se trata de una petición de escritura. Por tanto, las colas de lectura y de escritura almacenan una lista de peticiones en el orden en que éstas se hicieron. Asociado con cada petición hay un tiempo de expiración, con un valor por defecto de 0,5 segundos en caso de una petición de lectura y de 5 segundos en el de una escritura. Generalmente, el planificador extrae peticiones de la cola ordenada. Cuando se completa una petición, se elimina de la cabeza de la cola ordenada y también de la cola FIFO correspondiente. Sin embargo, cuando se cumple el tiempo de expiración del elemento de la cabeza de una de las colas FIFO, el planificador pasa a dar servicio de esa cola FIFO, extrayendo la petición expirada, junto con algunas de las siguientes peticiones de la cola. Según se sirve cada petición, se borra de la cola ordenada.

El esquema del planificador de E/S basado en plazos supera el problema de la inanición y también el problema de las lecturas frente a las escrituras.

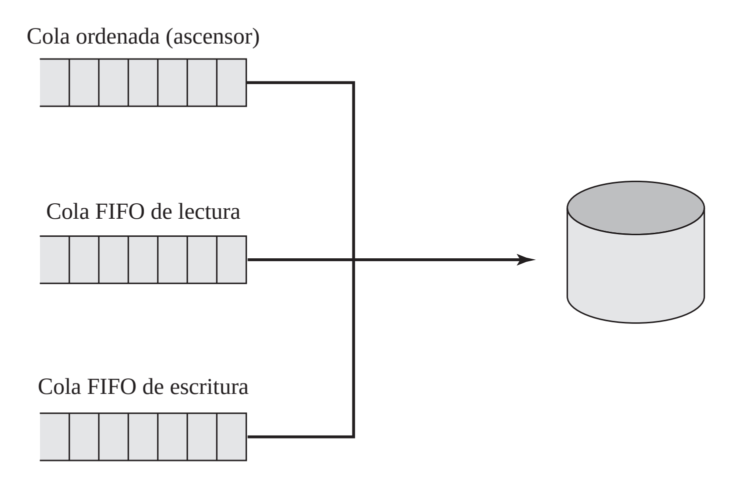


Figura 11.14. El planificador de E/S basado en plazos de Linux.

PLANIFICADOR DE E/S PREVISOR

El planificador del ascensor original y el basado en plazos están diseñados para servir una nueva petición tan pronto como se completa la petición existente, manteniendo, por tanto, el disco lo más ocupado que sea posible. Esta misma política se aplica a todos los algoritmos de planificación estudiados en la Sección 11.5. Sin embargo, esta política puede ser contraproducente si hay numerosas peticiones de lectura síncronas. Normalmente, una aplicación esperará hasta que se complete una petición de lectura y estén los datos disponibles antes de realizar la siguiente petición. El pequeño retardo que hay entre que se reciben los datos de la última lectura y la solicitud de la siguiente lectura permite al planificador dedicarse a otra petición pendiente y servir esa petición.

Gracias al principio de la proximidad, es probable que las lecturas sucesivas del mismo proceso se encuentren en bloques de disco que estén los unos cerca de los otros. Si el planificador tuviera un retardo de un breve periodo de tiempo después de servir una petición de lectura, de manera que pudiera comprobar si se hace una nueva petición de lectura cercana, el rendimiento global del sistema podría mejorarse. Ésta es la filosofía en la que se basa el planificador previsor, propuesto en [IYER01], e implementado en Linux 2.6.

En Linux, el planificador previsor está superpuesto sobre el planificador basado en plazos. Cuando se sirve una petición de lectura, el planificador previsor causa que el sistema de planificación se retrase hasta 6 milisegundos, dependiendo de la configuración. Durante este pequeño retardo, hay una oportunidad apreciable de que la aplicación que solicitó la última petición de lectura genere otra petición de lectura en la misma región del disco. En caso de que sea así, esa petición se servirá inmediatamente. Si no se produce esa petición de lectura, el planificador continúa utilizando el algoritmo de planificación basado en plazos.

[LOVE04b] muestra los resultados de dos pruebas del algoritmo de planificación de Linux. La primera prueba consistió en la lectura de un fichero de 200 MB mientras se hace una larga escritura secuencial ejecutando en segundo plano. En la segunda prueba se realizó una lectura de un fichero grande ejecutando en segundo plano mientras que se leen todos los ficheros del árbol de código fuente del núcleo. En la siguiente tabla se muestran los resultados de las pruebas:

Planificador de E/S y núcleo	Prueba 1	Prueba 2
Ascensor de Linus en 2.4	45 segundos	30 minutos y 28 segundos
Planificador de E/S basado en plazos en 2.6	40 segundos	3 minutos y 30 segundos
Planificador de E/S previsor en 2.6	4,6 segundos	15 segundos

Como se puede apreciar, la mejora del rendimiento depende de la naturaleza de la carga de trabajo. Sin embargo, en ambos casos, el planificador previsor proporciona una mejora muy considerable.

CACHE DE PÁGINAS DE LINUX

En Linux 2.2 y en las versiones anteriores, el núcleo mantiene una cache de páginas para las lecturas y escrituras de los ficheros ordinarios del sistema de ficheros y para las páginas de memoria virtual, y una cache de *buffers* independiente para la E/S de bloques. En Linux 2.4 y en las versiones posteriores, hay una única cache de páginas unificada que está involucrada en todo el tráfico entre el disco y la memoria principal.

La cache de páginas conlleva dos beneficios. En primer lugar, cuando llega el momento de escribir en el disco las páginas modificadas, se puede agrupar un conjunto de las mismas ordenándolas

adecuadamente y escribiéndolas, por tanto, eficientemente. En segundo lugar, gracias al principio de la proximidad temporal, las páginas incluidas en la cache de páginas se accederán probablemente de nuevo antes de ser expulsadas de la cache, evitando de esta forma una operación de E/S de disco.

Las páginas modificadas se escriben en el disco en dos situaciones:

- Cuando la cantidad de memoria libre llega a ser menor que un determinado umbral, el núcleo reduce el tamaño de la cache de páginas liberando memoria que va a añadirse al conjunto de memoria libre disponible en el sistema.
- Cuando las páginas modificadas envejecen más allá de un determinado umbral, se escriben en el disco varias páginas modificadas.

11.10. E/S DE WINDOWS

La Figura 11.15 muestra el gestor de E/S de Windows. Este gestor es responsable de todo el sistema de E/S del sistema operativo y proporciona una interfaz uniforme a la que todos los tipos de manejadores pueden llamar.

MÓDULOS DE E/S BÁSICOS

El gestor de E/S consta de cuatro módulos:

- **Gestor de cache.** El gestor de cache maneja la gestión de la cache para todo el subsistema de E/S, proporcionando un servicio de cache en memoria principal para todos los componentes de sistemas de ficheros y de red. Se puede incrementar y decrementar dinámicamente el tamaño de la cache dedicada a una determinada actividad según varíe la cantidad de la memoria física disponible. El gestor de cache incluye dos servicios para mejorar el rendimiento general:

  — **Escritura perezosa.** El sistema registra las actualizaciones sólo en la cache y no en el disco. Posteriormente, cuando el grado de utilización del procesador es bajo, el gestor de cache escribe los cambios en el disco. Si se actualiza un determinado bloque de cache mientras tanto, hay un ahorro neto.

  — **Compromiso perezoso.** Este servicio es similar a la escritura perezosa pero para procesamiento de transacciones. En vez de registrar de manera inmediata que una transacción se ha completado con éxito, el sistema almacena en la cache la información comprometida y, posteriormente, un proceso ejecutando en segundo plano la escribe en el *log* del sistema de ficheros.



Figura 11.15. Gestor de E/S de Windows.

- **Manejadores de sistemas de ficheros.** El gestor de E/S trata a un manejador de sistema de ficheros de la misma manera que a otro manejador y encamina los mensajes destinados a determinados volúmenes al manejador software correspondiente a ese adaptador de dispositivo.
- **Manejadores de red.** Windows incluye una gestión de red integrada y proporciona soporte para aplicaciones distribuidas.
- **Manejadores de dispositivos hardware.** Estos manejadores acceden a los registros hardware de los dispositivos periféricos a través de puntos de entrada de bibliotecas dinámicamente enlazadas del Ejecutivo de Windows. Existe un conjunto de estas rutinas para cada plataforma en la que puede ejecutar Windows; gracias a que los nombres de las rutinas son los mismos para todas las plataformas, el código fuente de los manejadores de dispositivos de Windows se puede transportar a distintos tipos de procesadores.

E/S SÍNCRONA Y ASÍNCRONA

Windows ofrece dos modos de operación para la E/S: asíncrono y síncrono. El modo asíncrono se utiliza para optimizar el rendimiento de la aplicación, siempre que sea posible. Con la E/S asíncrona, una aplicación inicia una operación de E/S prosiguiendo su ejecución mientras se lleva a cabo la petición de E/S. Con la E/S síncrona, la aplicación se bloquea hasta que se completa la operación de E/S.

La E/S asíncrona es más eficiente, desde el punto de vista del hilo que solicita la operación, ya que le permite continuar ejecutando mientras que el gestor de E/S encola la operación de E/S y, posteriormente, se realiza dicha operación. Sin embargo, la aplicación que invoca la operación de E/S asíncrona necesita alguna manera de determinar cuándo se completa la operación. Windows proporciona cuatro técnicas diferentes para la notificación de la finalización de una operación de E/S:

- **Activación de un objeto dispositivo del núcleo.** Con esta estrategia, se activa un indicador asociado a un objeto dispositivo cuando se completa una operación en ese objeto. El hilo que solicitó la operación de E/S puede continuar ejecutando hasta que alcance un punto donde debe parar a la espera de que se complete la operación de E/S. En ese instante, el hilo puede esperar hasta que se complete la operación y, a continuación, continuar su ejecución. Esta técnica es sencilla y fácil de utilizar pero no es apropiada para manejar múltiples peticiones de E/S. Por ejemplo, si un hilo necesita realizar múltiples acciones simultáneas sobre un único fichero, como leer un fragmento del fichero y escribir en otra parte del mismo, con esta técnica, el hilo no puede distinguir entre la finalización de la lectura y el de la escritura. Sabría simplemente que alguna operación de E/S solicitada sobre este fichero se ha completado.
- **Activación de un objeto evento del núcleo.** Esta técnica permite que haya múltiples peticiones de E/S simultáneas sobre un dispositivo o fichero. El hilo crea un evento por cada petición. Posteriormente, el hilo puede esperar por una sola de estas peticiones o por todas ellas.
- **E/S con alerta.** Esta técnica utiliza una cola asociada a un hilo, conocida como la cola de llamadas a procedimientos asíncronos (*Asynchronous Procedure Call*, APC). En este caso, el hilo realiza peticiones de E/S y el gestor de E/S sitúa los resultados de estas peticiones en la cola APC del hilo solicitante.
- **Puertos de finalización de E/S.** Esta técnica se utiliza en un servidor de Windows para optimizar el uso de los hilos. Esencialmente, consiste en que está disponible un conjunto de hilos para su uso, de manera que no es necesario crear un nuevo hilo para manejar una nueva petición.

RAID SOFTWARE

Windows proporciona dos tipos de configuraciones RAID, definidas en [MS96] de la siguiente forma:

- **RAID hardware.** Discos físicos independientes agrupados en uno o más discos lógicos por el hardware del controlador de disco o del armario de almacenamiento de disco.
- **RAID software.** Espacio de disco que no es contiguo agrupado en una o más particiones lógicas por el manejador de discos software tolerante a fallos, FTDISK.

En el RAID hardware, la interfaz del controlador maneja la creación y la regeneración de la información redundante. El RAID software, disponible en Windows Server, implementa la funcionalidad RAID como parte del sistema operativo y puede utilizarse con cualquier conjunto de múltiples discos. El sistema de RAID software implementa RAID 1 y RAID 5. En el caso del RAID 1 (duplicado de discos), los dos discos que contienen las particiones primaria y duplicada pueden estar asociados al mismo controlador de disco o a diferentes. A esta última configuración se la denomina *duplexing de disco*.

11.11. RESUMEN

La interfaz de un computador al mundo exterior corresponde con su arquitectura de E/S. Esta arquitectura está diseñada para proporcionar un medio sistemático de controlar la interacción con el mundo exterior y proveer al sistema operativo de la información que necesita para gestionar eficientemente la actividad de E/S.

El sistema de E/S se divide generalmente en varios niveles, de forma que los niveles inferiores tratan con los detalles que están más cercanos a las funciones físicas que se van a realizar y los niveles superiores tratan con la E/S de una manera lógica y genérica. Como resultado, los cambios en los parámetros del hardware no van a afectar a la mayoría del software de E/S.

Un aspecto fundamental de la E/S es la utilización de *buffers* que gestiona el sistema de E/S en lugar de los procesos de aplicación. El uso de *buffers* amortigua las diferencias entre la velocidad interna del computador y la velocidad de los dispositivos de E/S. El uso de *buffers* también desvincula la transferencia real de E/S del espacio de direcciones del proceso de aplicación. Esto permite al sistema operativo más flexibilidad a la hora de realizar sus funciones de gestión de memoria.

El aspecto de la E/S que tiene un mayor impacto en el rendimiento general del sistema es la E/S de disco. Por ello, ha habido un mayor auge de la investigación y el diseño en esta área que en otros tipos de E/S. La planificación de disco y la cache de disco constituyen dos de las estrategias más frecuentemente utilizadas para mejorar el rendimiento de E/S del disco.

En cualquier momento, puede haber una cola de peticiones de E/S en el mismo disco. El objetivo de la planificación del disco es satisfacer estas peticiones de manera que se minimice el tiempo de búsqueda mecánica del disco y, con ello, se mejore el rendimiento. En esta planificación entran en juego aspectos tales como la distribución física de las peticiones pendientes, así como consideraciones sobre la proximidad.

Una cache de disco es un *buffer*, almacenado usualmente en la memoria principal, que funciona como una cache de bloques de disco entre la memoria de disco y el resto de la memoria principal. Gracias al principio de la proximidad, el uso de una cache de disco debería reducir considerablemente el número de transferencias de E/S de bloques entre la memoria principal y el disco.

11.12. LECTURAS Y SITIOS WEB RECOMENDADOS

En la mayoría de los libros de arquitectura de computadores se pueden encontrar estudios generales de la E/S del computador, como [STAL03] y [PATT98]. [MEE96a] proporciona un buen estudio de la tecnología de grabación de los discos y las cintas. [MEE96b] se centra en las técnicas de almacena-

miento de datos en los sistemas de disco y cinta. [WIED87] contiene un excelente estudio de los aspectos relacionados con el rendimiento de disco, incluyendo aquéllos vinculados con la planificación del disco. [NG98] examina los aspectos de rendimiento del hardware del disco. [CAO96] analiza el uso de la cache de disco y la planificación de disco. [WORT94] y [SELT90] son buenos estudios de algoritmos de planificación de disco, que incluyen análisis del rendimiento.

[ROSC03] proporciona un completo resumen de todos los tipos de sistemas de memoria externa, incluyendo una moderada cantidad de detalles técnicos de cada uno. Otro interesante estudio, con mayor énfasis en la interfaz de E/S y menos en los dispositivos en sí mismos, es [SCHW96]. [PAI00] es una descripción pedagógica sobre un esquema integrado de gestión de *buffers* y cache en el sistema operativo.

[DELL00] proporciona un estudio detallado de los manejadores de dispositivos de Windows junto con una profunda revisión de toda la arquitectura de E/S de Windows.

Un excelente estudio de la tecnología RAID, escrita por los inventores del concepto de RAID, es [CHEN94]. En [MASS97] se presenta un estudio más detallado de la *RAID Advisory Board*, una asociación de distribuidores y consumidores de productos relacionados con RAID. [CHEN96] analiza el rendimiento del esquema RAID. Otro interesante artículo es [FRIE96]. [DALT96] describe en detalle el sistema de RAID software de Windows NT.

CAO96 Cao, P.; Felten, E.; Karlin, A.; y Li, K. «Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling.» *ACM Transactions on Computer Systems*, Noviembre 1996.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; y Patterson, D. «RAID: High-Performance, Reliable Secondary Storage.» *ACM Computing Surveys*, Junio 1994.

CHEN96 Chen, S., y Towsley, D. «A Performance Evaluation of RAID Architectures.» *IEEE Transactions on Computers*, Octubre 1996.

DALT96 Dalton, W., *et al. Windows NT Server 4: Security, Troubleshooting, and Optimization*. Indianapolis, IN: New Riders Publishing, 1996.

DELL00 Dekker, E., y Newcomer, J. *Developing Windows NT Device Drivers: A Programmer's Handbook*. Reading, MA: Addison Wesley, 2000.

FRIE96 Friedman, M. «RAID Keeps Going and Going and ...» *IEEE Spectrum*, Abril 1996.

MASS97 Massiglia, P. (editor). *The RAID Book: A Storage System Technology Handbook*. St. Peter, MN: The Raid Advisory Board, 1997.

MEE96A Mee, C., y Daniel, E. eds. *Magnetic Recording Technology*. New York: McGraw-Hill, 1996.

MEE96B Mee, C., y Daniel, E. eds. *Magnetic Storage Handbook*. New York: McGraw-Hill, 1996.

NG98 Ng, S. «Advances in Disk Technology: Performance Issues». *Computer*, Mayo 1989.

PAI00 Pai, V.; Druschel, P.; y Zwaenepoel, W. «IO-Lite: A Unified I/O Buffering and Caching System». *ACM Transactions on Computer Systems*, Febrero 2000.

PATT98 Patterson, D., y Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, CA: Morgan Kaufmann, 1998.

ROSC03 Rosch, W. *The Winn L. Rosch Hardware Bible*. Indianapolis, IN: Sams, 2003.

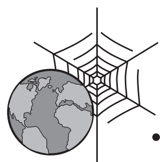
SCHW96 Schwaderer, W., y Wilson, A. *Understanding I/O Subsystems*. Milpitas, CA: Adaptec Press, 1996.

SELT90 Seltzer, M.; Chen, P.; y Ousterhout, J. «Disk Scheduling Revisited.» *Proceedings, USENIX Winter Technical Conference*, Enero 1990.

STAL03 Stallings, W. *Computer Organization and Architecture*, 5th ed. Upper Saddle River, NJ: Prentice Hall, 2000.

WIED87 Wiederhold, G. *File Organization for Database Design*. New York: McGraw-Hill, 1987.

WORT94 Worthington, B.; Ganger, G.; y Patt, Y. «Scheduling Algorithms for Modern Disk Drives.» *ACM SiGMETRICS*, Mayo 1994.



SITIOS WEB RECOMENDADOS

- **Caracterización y optimización de la E/S.** Un sitio dedicado a la educación e investigación en el área del diseño y del rendimiento de E/S. Incluye herramientas y cursos útiles. Gestionado por la universidad de Illinois.

11.13. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

acceso directo a memoria (DMA)	disco duro	paquete de discos
bloque	disco extraíble	pista
buffer circular	disco flexible	procesador de E/S
buffer de E/S	disco magnético	retardo rotacional
cabeza de lectura/escritura	disco no extraíble	sector
cache de disco	dispositivo orientado a bloques	tiempo de acceso al disco
canal de E/S	dispositivo orientado a flujo de caracteres	tiempo de búsqueda
CD-R	Entrada/salida (E/S)	tiempo de transferencia
CD-ROM	E/S de dispositivo	vector redundante de discos independientes (RAID)
CD-RW	E/S dirigida por interrupciones	
cilindro	E/S lógica	
disco de cabeza fija	E/S programada	
disco de cabeza móvil	hueco	
disco digital versátil (DVD)		

CUESTIONES DE REPASO

11.1. Cite ejemplos de recursos reutilizables y consumibles.

11.2. Enumere y defina brevemente tres técnicas para realizar E/S.

11.3. ¿Cuál es la diferencia entre la E/S lógica y la E/S de dispositivos?

11.4. ¿Cuál es la diferencia entre los dispositivos orientados a bloques y los orientados a flujos de caracteres? Proponga algunos ejemplos de cada tipo.

11.5. ¿Por qué se debería mejorar el rendimiento utilizando para E/S un *buffer* doble en lugar de un único *buffer*?

11.6. ¿Qué elementos de retardo están involucrados en una lectura o escritura de disco?

11.7. Defina brevemente las políticas de planificación de disco mostradas en la Figura 11.7.

11.8. Defina brevemente los siete niveles RAID.

11.9. ¿Cuál es el tamaño habitual del sector del disco?

PROBLEMAS

11.1. Considere un programa que accede a un único dispositivo de E/S y compare la E/S sin *buffer* con el uso de un *buffer*. Muestre que el uso del *buffer* puede reducir el tiempo de ejecución en un factor de dos como máximo.

11.2. Generalice el resultado del Problema 11.1 para el caso de un programa que utiliza *n* dispositivos.

11.3. Realice el mismo tipo de análisis que el que se muestra en la Tabla 11.2 para la siguiente secuencia de peticiones de pistas de disco: 27, 129, 110, 186, 147, 41, 10, 64 y 120. Asuma que la cabeza del disco está situada inicialmente en la pista 100 y se está moviendo en la dirección de números de pista crecientes. Repita el mismo análisis, pero ahora suponiendo que la cabeza del disco se está moviendo en la dirección contraria.

11.4. Considere un disco con N pistas numeradas de 0 a $(N-1)$ y suponga que los sectores pedidos se distribuyen aleatoria y uniformemente sobre el disco. Se pretende calcular el número medio de pistas atravesadas en una búsqueda.

a) En primer lugar, calcule la probabilidad de que se produzca una búsqueda de longitud j cuando la cabeza está actualmente situada sobre la pista t. *Sugerencia*: hay que determinar el número total de combinaciones, teniendo en cuenta que todas las posiciones de pista tienen la misma probabilidad de ser el destino de la búsqueda.

b) A continuación, calcule la probabilidad de que se produzca una búsqueda de longitud K. *Sugerencia:* Este cálculo implica la suma de todas las posibles combinaciones de movimientos de K pistas.

c) Calcule el número medio de pistas atravesadas en una búsqueda, usando la fórmula que permite obtener el valor esperado:

$$E[x] = \sum_{i=0}^{N-1} i \sum \Pr[x = i]$$

d) Demuestre que para valores grandes de N, el número medio de pistas atravesadas en una búsqueda es aproximadamente igual a $N/3$.

11.5. Se ha propuesto la siguiente ecuación tanto para la memoria cache como para la memoria cache de disco:

$$T_S = T_C + M \times T_D$$

Generalice esta ecuación para una jerarquía de memoria con N niveles en lugar de únicamente con 2.

11.6. Dado el algoritmo de remplazo basado en la frecuencia (*véase* la Figura 11.11), se define F_{nueva}, $F_{intermedia}$ y $F_{antigua}$ como la fracción de la cache que corresponde con las secciones nueva, intermedia y antigua, respectivamente. Evidentemente, se cumplirá que $F_{nueva} + F_{intermedia} + F_{antigua} = 1$. Analice de qué tipo de política se trata si se cumple:

a) $F_{antigua} = 1 - F_{nueva}$

b) $F_{antigua} = 1$ / (tamaño de la cache)

11.7. ¿Cuál es la velocidad de transferencia de una unidad de cinta magnética de nueve pistas si la velocidad de cinta es de 120 pulgadas por segundo y la densidad de la misma es de 1.600 bits lineales por pulgada?

11.8. Sea una bobina de cinta de 2.400 pies; y una zona de separación entre registros de 0,6 pulgadas, en donde la cinta se detiene entre dos lecturas; una aceleración/deceleración lineal durante los arranques/paradas en las zonas de separación entre registros; y las otras caracterísiticas de la cinta iguales a las del Problema 11.7. Los datos en la cinta están organizados en registros físicos, tal que cada registro físico contiene un número fijo de unidades definidas por el usuario, llamadas registros lógicos.

a) ¿Cuánto tiempo llevará leer una cinta completa de registros lógicos de 120 bytes agrupados en bloques de 10 por cada registro físico?

b) Lo mismo que en el Apartado (a), pero en este caso agrupados en bloques de 30.

c) ¿Cuántos registros lógicos almacenará la cinta usando cada uno de los dos factores de agrupamiento de bloques mencionados anteriormente?

d) ¿Cuál es la tasa de transferencia global efectiva para cada uno de los dos factores de agrupamiento de bloques mencionados anteriormente?

e) ¿Cuál es la capacidad de la cinta?

11.9. Calcule cuánto espacio de disco (en sectores, pistas y superficies) se requiere para almacenar los registros lógicos leídos en el Problema 11.8b si el disco tiene sectores de tamaño fijo de 512 bytes/sector, 96 sectores/pista, 110 pistas por superficie y 8 superficies útiles. Ignore cualquier tipo de registro (o registros) de cabecera de fichero o de índices de pista, y suponga que un registro no se puede extender sobre dos sectores.

11.10. Considere el sistema de disco descrito en el Problema 11.9 y suponga que el disco rota a 360 rpm. En este sistema, un procesador lee un sector del disco utilizando E/S dirigida por interrupciones, produciéndose una interrupción por cada byte. Si se tarda 2,5 µs en procesar cada interrupción, ¿cuál es el porcentaje de tiempo que el procesador dedica a manejar la E/S (sin tener en cuenta el tiempo de búsqueda)?

11.11. Repita el Problema 11.10 usando DMA y suponiendo una interrupción por sector.

11.12. Un computador de 32 bits tiene dos canales selectores y un canal multiplexor. Cada canal selector gestiona dos unidades de disco magnético y dos de cinta magnética. El canal multiplexor tiene conectados dos impresoras de líneas, dos lectores de tarjetas y diez terminales VDT. Suponga las siguientes velocidades de transferencia:

Dispositivo de disco	800 Kbytes/s
Dispositivo de cinta magnética	200 Kbytes/s
Impresora de líneas	6,6 Kbytes/s
Lector de tarjetas	1,2 Kbytes/s
VDT	1 Kbytes/s

Estime la máxima velocidad de transferencia de E/S conjunta en este sistema.

11.13. Debería ser evidente que la distribución de datos en bandas en los discos puede mejorar la tasa de transferencia de datos cuando el tamaño de la banda es pequeño comparado con el tamaño de la petición de E/S. Asimismo, debería ser evidente que el esquema RAID 0 proporciona una mejora en el rendimiento comparándolo con un único disco grande, ya que pueden manejarse en paralelo múltiples peticiones de E/S. Sin embargo, en este último caso, ¿es necesaria la distribución de datos en bandas? Es decir, ¿el uso de esta técnica mejora la tasa de peticiones de E/S respecto a la proporcionada por un vector de discos de características similares pero que no utiliza dicha técnica?

APÉNDICE 11A DISPOSITIVOS DE ALMACENAMIENTO EN DISCO

DISCO MAGNÉTICO

Un disco es un plato circular construido de metal o plástico que está cubierto con un material magnético. Los datos se graban en el disco y posteriormente se recuperan del mismo mediante una bobina conductora llamada **cabeza**. Durante una operación de lectura o escritura, la cabeza está estacionaria mientras que el plato rota debajo de ella.

El mecanismo de escritura se basa en el hecho de que cuando la electricidad fluye a través de una bobina se produce un campo magnético. Se envían pulsos a la cabeza, registrándose patrones magnéticos en la superficie subyacente, que serán diferentes dependiendo de si la corriente es positiva o negativa. El mecanismo de lectura se basa en que un campo magnético moviéndose con respecto a una bobina produce una corriente eléctrica en la bobina. Cuando la superficie del disco pasa debajo de la cabeza, se genera una corriente de la misma polaridad que la que se grabó previamente.

ORGANIZACIÓN Y FORMATO DE LOS DATOS

La cabeza es un dispositivo relativamente pequeño capaz de leer y escribir sobre una parte del plato que rota debajo de la misma. En consecuencia, la organización de datos en el plato consiste en un conjunto concéntrico de anillos, llamados **pistas**. Cada pista tiene la misma anchura que la cabeza. Hay miles de pistas en cada superficie.

La Figura 11.16 muestra esta distribución de datos. Las pistas adyacentes están separadas por **huecos**. Esto impide, o al menos minimiza, los errores debidos a que la cabeza no esté alineada o, simplemente, a que existan interferencias en los campos magnéticos.

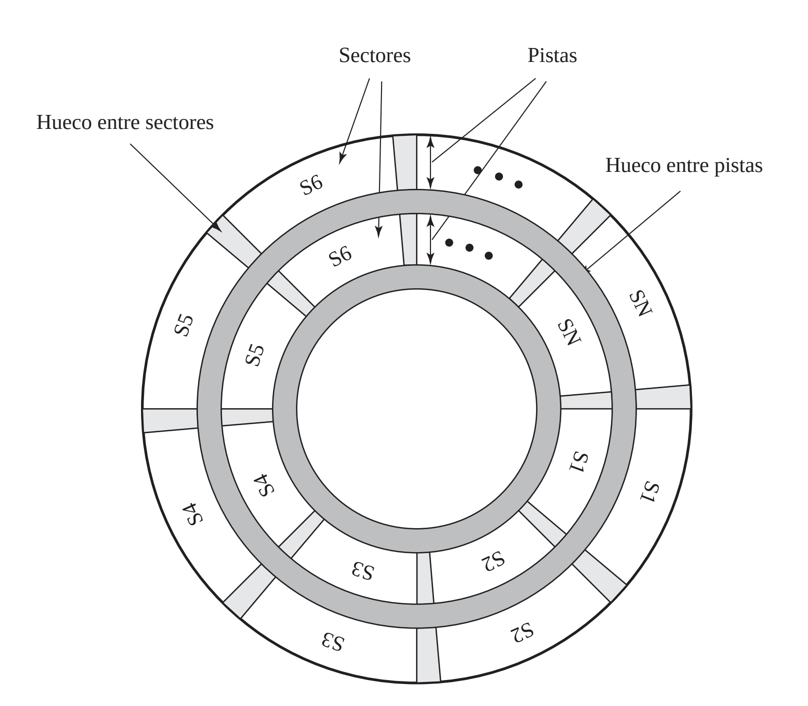
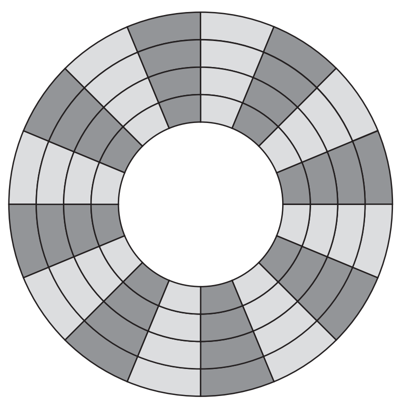


Figura 11.16. Disposición de los datos en el disco.

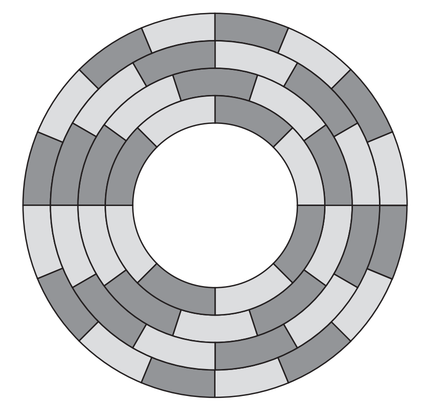
Los datos se transfieren del disco en **sectores** (Figura 11.16). Normalmente, hay cientos de sectores por pista, que pueden ser de longitud fija o variable. En la mayoría de los sistemas contemporáneos, los sectores utilizan una longitud fija de 512 bytes, siendo prácticamente universal el tamaño del sector. Para evitar imponer requisitos de precisión irrazonables en el sistema, los sectores adyacentes están separados por huecos entre las pistas, además de existir huecos entre los sectores de la misma pista.

Un bit que esté próximo al centro de un disco rotando pasa a través de un punto fijo (como una cabeza de lectura-escritura) a menos velocidad que un bit situado en la parte exterior. Por tanto, se debe encontrar alguna manera de compensar esta variación en la velocidad de manera que la cabeza pueda leer todos los bits a la misma velocidad. Esto se puede lograr incrementando el espacio entre los bits de información grabada en distintos segmentos del disco. Gracias a ello, la información puede accederse a la misma velocidad rotando el disco a una velocidad fija, conocida como **velocidad angular constante (*Constant Angular Velocity*, CAV)**. La Figura 11.17a muestra la disposición de un disco que utiliza **CAV**. El disco está dividido en varios sectores en forma de tarta y en una serie de pistas concéntricas. La ventaja de utilizar CAV es que se puede hacer referencia directamente a cada bloque individual de datos usando un número de pista y sector. Para mover la cabeza desde su posición actual a una dirección específica, sólo se necesita un corto movimiento de la cabeza a la pista correspondiente y una breve espera hasta que el sector apropiado gire debajo de la cabeza. La desventaja de CAV es que la cantidad de datos que se pueden almacenar en las pistas exteriores, que tienen mayor longitud, es la misma que se puede almacenar en las pistas interiores, que son más cortas.

Dado que la **densidad**, en bits por pulgada lineal, se incrementa al moverse desde las pistas más externas hasta las más internas, la capacidad de almacenamiento del disco en un sistema CAV sencillo está limitada por la densidad de grabación máxima que puede lograrse en las pistas más internas. Para incrementar la densidad, los sistemas de disco duro modernos utilizan una técnica conocida como **grabación en múltiples zonas**, en la que la superficie está dividida en varias zonas concéntricas (normalmente, 16). Dentro de una zona, el número de bits por pista es constante. Las zonas más alejadas del centro contienen más bits (más sectores) que las zonas que están más cerca del centro. Esto permite una mayor capacidad de almacenamiento global con el coste de una circuitería algo más compleja. Según se mueve la cabeza del disco de una zona a otra, cambia la longitud (a lo largo de la pista) de cada bit, causando un cambio en la temporización de las lecturas y escrituras. La Figura 11.17b muestra la naturaleza de la grabación en múltiples zonas; en esta figura, cada zona tiene una anchura de una sola pista.



(a) Velocidad angular constante



(b) Grabación en múltiples zonas

Figura 11.17. Comparación de métodos de disposición del disco.

Se necesita alguna forma de localizar la posición de cada sector dentro de una pista. Lógicamente, debe de haber algún punto de inicio en la pista y una manera de identificar el inicio y el final de cada sector. Estos requisitos se manejan por medio de datos de control grabados en el disco. Por tanto, el disco se formatea con algunos datos adicionales utilizados sólo por el controlador de disco y que no están accesibles para el usuario.

CARACTERÍSTICAS FÍSICAS

La Tabla 11.6 detalla las características principales que distinguen los diversos tipos de discos magnéticos. En primer lugar, la cabeza puede estar fija o moverse con respecto a la dirección radial del plato. En un **disco con cabeza fija**, hay una cabeza de lectura/escritura por pista. Todas las cabezas están montadas en un brazo rígido que se extiende a lo largo de las pistas. En un **disco de cabeza móvil**, hay sólo una cabeza de lectura/escritura por cada superficie. También en este caso la cabeza está montada en un brazo. Dado que la cabeza debe de ser capaz de situarse sobre cualquier pista, el brazo puede extenderse o retraerse con este propósito.

Tabla 11.6. Características físicas de los sistemas de disco.

Movimiento de la cabeza	**Platos**
Cabeza fija (una por pista)	Único plato
Cabeza móvil (una por superficie)	Múltiples platos
Carácter portátil del disco	**Mecanismo de la cabeza**
Disco no extraíble	Contacto (disco flexible)
Disco extraíble	Separación fija
	Separación aerodinámica (Winchester)
Número de caras por plato	
Sólo una cara	
Doble cara	

El disco en sí mismo está montado en un dispositivo de disco, que consta de un brazo, un eje giratorio que rota el disco y la electrónica necesaria para leer y escribir datos binarios. Un **disco no extraíble** está montado permanentemente en el dispositivo de disco; el disco duro de un computador personal es un disco no extraíble. Un **disco extraíble** puede ser extraído y remplazado por otro disco. La ventaja de este tipo de discos es que permiten que esté disponible una cantidad ilimitada de datos con un número limitado de sistemas de disco. Además, un disco de este tipo se puede mover desde un computador a otro. Los discos flexibles y los cartuchos ZIP son ejemplos de discos extraíbles.

En la mayoría de los discos, la cubierta magnética está aplicada en ambos lados del plato, a lo que se denomina **doble cara**. Algunos discos más económicos utilizan discos de **una sola cara**.

Algunos dispositivos de disco incluyen **múltiples platos** apilados verticalmente separados por menos de una pulgada, existiendo múltiples brazos (Figura 11.18). Los discos con múltiples platos emplean una cabeza móvil, con una cabeza de lectura/escritura por cada superficie del plato. Todas las cabezas están fijas mecánicamente, de manera que todas están a la misma distancia del centro del disco y se mueven conjuntamente. Así, en todo momento, todas las cabezas se posicionan sobre pistas que están a la misma distancia del centro del disco. Al conjunto de todas las pistas en la misma posición relativa en el plato se le denomina **cilindro**. Por ejemplo, todas las pistas sombreadas en la Figura 11.19 son parte de un cilindro.

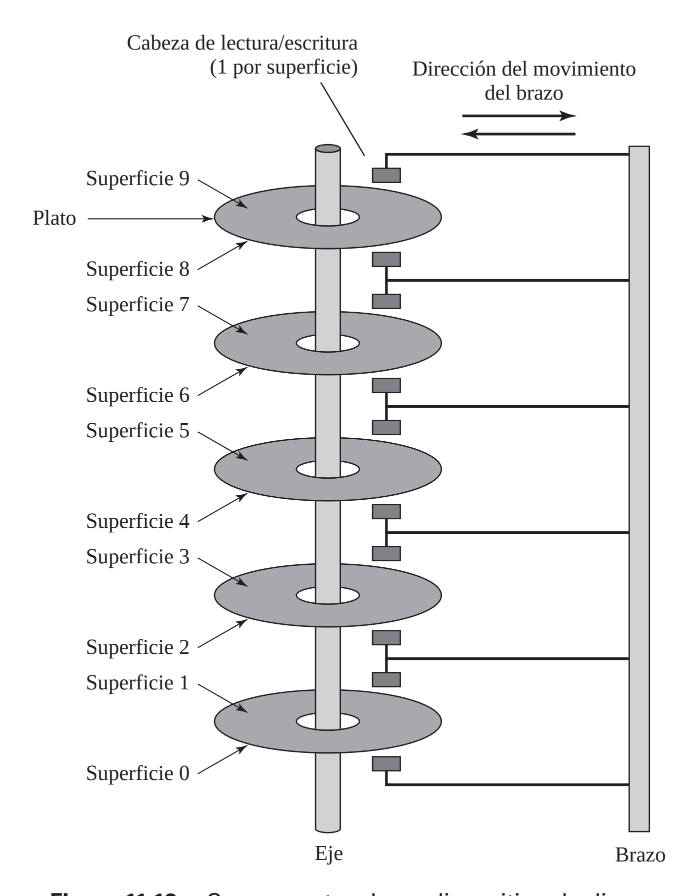


Figura 11.18. Componentes de un dispositivo de disco.

Por último, el mecanismo de la cabeza establece una clasificación de los discos en tres tipos. Tradicionalmente, la cabeza de lectura/escritura se posiciona a una distancia fija sobre el plato, dejando una bolsa de aire. En el otro extremo, existe un mecanismo de cabeza que entra realmente en contacto físico con el medio durante una operación de lectura/escritura. Este mecanismo se usa en el **disco flexible**, que es un plato pequeño y flexible, constituyendo el tipo de disco más económico.

Para comprender el tercer tipo de disco, se necesita comentar la relación existente entre la densidad de datos y el tamaño de la bolsa de aire. La cabeza debe generar o detectar un campo electromagnético de magnitud suficiente para leer y escribir apropiadamente. Cuanto más estrecha es la cabeza, más cerca debe estar de la superficie del plato para poder funcionar. Una cabeza más estrecha implica pistas más estrechas y, por tanto, mayor densidad de datos, lo que es deseable. Sin embargo, cuanto más cerca esté la cabeza del disco, mayor es el riesgo de error por impurezas e imperfecciones. El desarrollo del **Disco Winchester** significó un avance considerable en esta tecnología. Las cabezas Winchester se utilizan en montajes sellados de dispositivos que están prácticamente libres de contaminantes. Están diseñados para operar más cerca de la superficie del disco que las cabezas del disco rígido convencional, permitiendo una mayor densidad de datos. La cabeza es realmente una lámina aerodinámica que descansa ligeramente en la superficie del plato cuando el disco está inmóvil. La presión del aire generada por un disco cuando está girando es suficiente para hacer que la lámina se eleve por encima de la superficie. El sistema sin contactos resultante puede construirse de manera que se usen

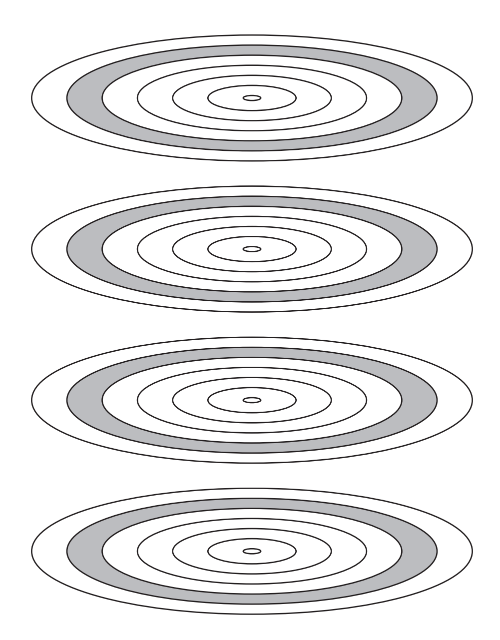


Figura 11.19. Pistas y cilindros.

cabezas más estrechas que operen más cerca de la superficie del plato que las cabezas de los disco duros convencionales[4].

La Tabla 11.7 muestra los parámetros de típicos discos de cabezas móviles de alto rendimiento actuales.

MEMORIA ÓPTICA

En 1983, se presentó uno de los productos de consumo de mayor éxito de todos los tiempos: el sistema de audio digital en disco compacto (*Compact Disk,* CD). El CD es un disco que no puede borrarse y que puede almacenar más de 60 minutos de información de audio en una cara. El enorme éxito comercial del CD permitió el desarrollo de la tecnología de almacenamiento de disco óptico de bajo coste que ha revolucionado el almacenamiento de datos del computador. Desde entonces, se han presentado diversos sistemas de disco óptico (*véase* la Tabla 11.8). A continuación, se presenta brevemente cada uno de ellos.

---

[4] Como asunto de interés historico, el término Winchester lo utilizó originalmente IBM como un nombre de código para el modelo de disco 3340 antes de ser anunciado públicamente. El 3340 era un paquete de discos extraíble con las cabezas selladas dentro del paquete. El término se aplica hoy en día a cualquier dispositivo de disco cuyas unidades están selladas y que usa un diseño de cabeza aerodinámico. El disco Winchester se usa comúnmente en computadores personales y estaciones de trabajo, donde se le llama **disco duro**.

Tabla 11.7. Parámetros de algunos discos duros típicos.

Características	Seagate Barracuda 180	Seagate Cheetah X15-36LP	Seagate Barracuda 36ES	Toshiba HDD1242	IBM Microdrive
Aplicación	Servidor de alta capacidad	Servidor de alto rendimiento	Computador personal de gama baja	Computador portátil	Dispositivos portátiles de mano
Capacidad	181,6 GB	36,7 GB	18,4 GB	5 GB	1 GB
Tiempo mínimo de búsqueda de pista a pista	0,8 ms	0,3 ms	1,0 ms	—	1,0 ms
Tiempo medio de búsqueda	7,4 ms	3,6 ms	9,5 ms	15 ms	12 ms
Velocidad de rotación	7.200 rpm	15K rpm	7.200 rpm	4.200 rpm	3.600 rpm
Retardo rotacional medio	4,17 ms	2 ms	4,17 ms	7,14 ms	8,33 ms
Tasa de transferencia máxima	160 MB/ s	522 a 709 MB/s	25 MB/s	66 MB/s	13,3 MB/s
Bytes por sector	512	512	512	512	512
Sectores por pista	793	485	600	63	—
Pistas por cilindro (número de caras de los platos)	24	8	2	2	2
Cilindros (número de pistas en una cara de un plato)	24.247	18.479	29.851	10.350	—

Tabla 11.8. Diversos tipos de discos ópticos.

CD	Disco compacto. Un disco que no puede borrarse y que almacena información de audio digitalizada. El sistema estándar usa discos de 12 cm. y puede grabar más de 60 minutos de tiempo de reproducción ininterrumpida.
CD-ROM	Disco compacto de memoria de sólo lectura (*Read Only Memory*). Un disco que no puede borrarse, utilizado para almacenar datos del computador. El sistema estándar usa discos de 12 cm. y puede almacenar más de 650 Mbytes.
CD-R	CD grabable (*Recordable*). Similar a un CD-ROM. El usuario sólo puede escribir una vez en el disco.
CD-RW	CD modificable (*Rewritable*). Similar a un CD-ROM. El usuario puede borrar y modificar el disco múltiples veces.
DVD	Disco Digital Versátil (*Digital Versatile Disk*). Una tecnología para producir una representación comprimida y digitalizada de información de vídeo, así como de grandes volúmenes de otros datos digitales. Se usan discos de 8 y de 12 cm. de diámetro, con una capacidad de hasta 17 Gbytes usando doble cara. El DVD básico es de sólo lectura (DVD-ROM).
DVD-R	DVD grabable (*Recordable*). Similar a un DVD-ROM. El usuario sólo puede escribir una vez en el disco. Sólo se pueden usar discos de una única cara.
DVD-RW	DVD modificable (*Rewritable*). Similar a un DVD-ROM. El usuario puede borrar y modificar el disco múltiples veces. Sólo se pueden usar discos de una única cara.

CD-ROM

Tanto el CD de audio como el CD-ROM (*Compact Disk Read Only Memory*, disco compacto de sólo lectura) usan una tecnología similar. La diferencia principal es que los reproductores de CD-ROM son más robustos y tienen mecanismos de corrección de errores que aseguran que los datos se transfieren correctamente desde el disco al computador. Ambos tipos de discos se fabrican de la misma manera. El disco está hecho de una resina, como, por ejemplo, policarbonato. La información grabada digitalmente (ya sea música o datos del computador) se imprime como una serie de agujeros microscópicos en la superficie del policarbonato. Esta operación se realiza, en primer lugar, con un láser de alta intensidad enfocado con precisión, creando un disco maestro. El maestro se usa, a su vez, para hacer una matriz que permita imprimir copias en policarbonato. A continuación, la superficie agujereada se cubre con una superficie muy reflectante, normalmente de aluminio o de oro. Esta superficie brillante está protegida del polvo y de los rasguños por una cubierta superior de material acrílico transparente. Por último, se puede realizar una serigrafía de una etiqueta sobre el material acrílico.

La información se lee de un CD o de un CD-ROM mediante un láser de baja intensidad alojado en el reproductor de disco óptico, o unidad del dispositivo. El láser ilumina a través de la cubierta protectora transparente mientras que un motor gira el disco que va pasando a través del mismo. La in-

tensidad de la luz del láser reflejada cambia cuando enfoca a un agujero. Este cambio lo detecta un foto sensor y lo convierte en una señal digital.

Recuerde que en un disco magnético la información se graba en pistas concéntricas. Si se usa en el disco magnético un sistema de velocidad angular constante (*Constant Angular Velocity*, CAV), que es más sencillo, el número de bits por pista es constante. Para lograr un incremento de la densidad, se usa un esquema de grabación en múltiples zonas, en el que la superficie está dividida en varias zonas, de manera que las más alejadas del centro contienen más bits que las más cercanas al mismo. Aunque esta técnica incrementa la capacidad, sigue sin ser óptima.

Para alcanzar mayor capacidad, el CD y el CD-ROM no organizan la información en pistas concéntricas. En su lugar, el disco contiene una única pista en espiral, comenzando cerca del centro y girando en forma de espiral hacia el extremo exterior del disco. Los sectores cercanos al exterior del disco tienen la misma longitud que los que están cerca del interior. Por tanto, la información está empaquetada uniformemente a lo largo del disco en segmentos del mismo tamaño, que se acceden a la misma velocidad rotando el disco a una velocidad variable. El láser lee los agujeros a una **velocidad lineal constante** (***Constant Linear Velocity*, CLV**). El disco rota más lentamente cuando se accede al extremo más exterior que cuando se accede cerca del centro. Por tanto, de igual manera la capacidad de una pista como el retardo rotacional se incrementan para las posiciones más cercanas al extremo más exterior del disco. La capacidad del CD-ROM es aproximadamente de 680 MB.

El CD-ROM es apropiado para la distribución de grandes cantidades de datos a un gran número de usuarios. Sin embargo, debido al gasto generado en el proceso inicial de escritura, no es apropiado para una sola aplicación. Comparándolo con los discos magnéticos tradicionales, el CD-ROM tiene tres ventajas principales:

- La capacidad de almacenamiento de información es mayor en el disco óptico.
- El disco óptico junto con la información almacenada en el mismo puede ser replicada masivamente de forma económica, a diferencia de lo que ocurre con un disco magnético. Para reproducir la base de datos almacenada en un disco magnético hay que copiarla disco a disco utilizando dos unidades de disco.
- El disco óptico es extraíble, permitiendo utilizar el disco en sí mismo como almacenamiento para el archivo permanente de información. La mayoría de los discos magnéticos no son extraíbles. En los discos magnéticos que no son extraíbles la información debe copiarse primero en cinta antes de que el dispositivo de disco, y el propio disco, pueda utilizarse para almacenar nueva información.

Las desventajas del CD-ROM son las siguientes:

- Es sólo de lectura y no puede actualizarse.
- Tiene un tiempo de acceso mucho mayor que el dispositivo de disco magnético, tanto como medio segundo.

CD GRABABLE

Para adaptarse a las aplicaciones en las que sólo se necesita una copia, o un pequeño número de copias, de un conjunto de datos, se ha desarrollado el CD que permite muchas lecturas pero sólo una escritura, que se denomina CD grabable (CD-R, CD *Recordable*). En un CD-R, se fabrica un disco de modo que posteriormente se puede escribir una vez con un rayo láser de moderada intensidad. Por

tanto, con un controlador de disco un poco más caro que el de CD-ROM, además de leer el disco, el consumidor puede escribir una vez en el mismo.

El substrato del CD-R es similar, aunque no idéntico, al CD o CD-ROM. En el caso del CD y el CD-ROM, la información se graba mediante agujeros en la superficie del medio, que cambia su capacidad reflectante. En un CD-R, el substrato incluye una capa con tinte. El tinte se utiliza para cambiar la capacidad reflectante, siendo activado por un láser de alta intensidad. El disco resultante se puede leer en un dispositivo de CD-R o en uno de CD-ROM.

El disco óptico CD-R está indicado para el archivo permanente de documentos y ficheros. Proporciona un registro permanente de grandes volúmenes de datos de usuario.

CD MODIFICABLE

El disco óptico CD-RW (*Rewritable*) se puede escribir y reescribir repetidas veces, como ocurre con un disco magnético. Aunque se han intentado usar diversas estrategias, la única técnica de carácter puramente óptico (en contraposición con la técnica magneto-óptica, que se presenta posteriormente) que se ha mostrado efectiva se denomina cambio de fase. El disco de cambio de fase utiliza un material que presenta dos niveles reflectantes significativamente diferentes en dos estados de fase distintos. Hay un estado amorfo, en el que las moléculas presentan una orientación aleatoria que apenas refleja la luz, y un estado cristalino que tiene una superficie lisa que refleja bien la luz. Un rayo de luz láser puede cambiar el material de una fase a otra. La desventaja primordial de los discos ópticos de cambio de fase es que el material acaba perdiendo de forma definitiva y permanente sus propiedades necesarias. Los materiales actuales se pueden utilizar durante un número de ciclos de borrado que van desde los 500.000 hasta 1.000.000.

El CD-RW tiene una ventaja evidente con respecto al CD-ROM y al CD-R: puede reescribirse y, por tanto, usarse como un verdadero almacenamiento secundario. Por ello, compite con el disco magnético. Una ventaja fundamental de los discos ópticos es que los márgenes de tolerancia en su fabricación son mucho menos estrictos que en los discos magnéticos de alta capacidad. Por tanto, presentan una mayor fiabilidad y una vida más prolongada.

DISCO DIGITAL VERSÁTIL

Con la gran capacidad del disco digital versátil (*Digital Versatile Disk*, DVD), la industria electrónica ha encontrado por fin un sustituto aceptable de la cinta analógica de vídeo VHS. El DVD remplazará a la cinta de vídeo utilizada en los grabadores de casetes de vídeo (*Video Cassette Recorder*, VCR) y, más importante en el ámbito de esta presentación, remplazará al CD-ROM en los computadores personales y servidores. EL DVD introduce al vídeo en la era digital. Muestra películas con una calidad de imagen impresionante, pudiendo accederse de manera aleatoria como en el CD audio, que también puede reproducirse en las máquinas DVD. En el disco DVD se pueden almacenar enormes volúmenes de datos: actualmente siete veces más que en un CD-ROM. Con la enorme capacidad de almacenamiento del DVD y su increíble calidad, los juegos de PC se harán más realistas y el software pedagógico incorporará un mayor uso del vídeo. Junto con el despertar de esta tecnología, se producirá una nueva explosión en el tráfico en Internet y en las intranets corporativas, según se vaya incorporando este material en los sitios web. La mayor capacidad del DVD se debe a las siguientes tres diferencias con respecto al CD-ROM:

- En el DVD los bits se almacenan más próximos entre sí. El espacio entre los bucles de una espiral en un CD es de 1,6 μm y la distancia mínima entre los agujeros a lo largo de la espiral es de 0,834 μm. El DVD utiliza un láser con una longitud de onda más corta alcanzando un espa-

cio de bucle de 0,74 µm y una distancia mínima entre agujeros de 0,4 µm. El resultado de estas dos mejoras es un incremento en la capacidad de aproximadamente siete veces la del CD-ROM, hasta cerca de los 4,7 GB.

- El DVD emplea una segunda capa de agujeros sobre un substrato encima de la primera capa. Un DVD de capa dual tiene una capa semi-reflectora encima de la capa reflectora, de manera que ajustando el foco, los lectores láser de los dispositivos DVD pueden leer cada capa separadamente. Esta técnica casi dobla la capacidad del disco, hasta casi 8,5 GB. La menor capacidad reflectora de la segunda capa limita su capacidad de almacenamiento por lo que no se llega al doble de capacidad.
- El DVD puede tener dos caras, mientras que en el CD los datos se graban en una sola cara, lo que proporciona una capacidad total de 17 GB.

Como ocurre con el CD, el DVD se presenta en versiones modificables, así como de sólo lectura (Tabla 11.8).

CAPÍTULO 12

Gestión de ficheros

◆ ◆

En la mayoría de las aplicaciones, el fichero es el elemento central. Con la excepción de las aplicaciones de tiempo real y algunas aplicaciones especializadas, la entrada a la aplicación se realiza mediante un fichero y en prácticamente todas las aplicaciones, la salida se guarda en un fichero para un almacenamiento a largo plazo o para su acceso posterior por parte del usuario u otros programas.

Los ficheros tienen vida fuera de cualquier aplicación individual que los utilice como entrada y/o salida. Los usuarios desean poder acceder a los ficheros, guardarlos y mantener la integridad de sus contenidos. Con el fin de lograr estos objetivos, prácticamente todos los sistemas operativos proporcionan programas que se ejecutan como aplicaciones privilegiadas. Sin embargo, como mínimo, un sistema de gestión de ficheros necesita servicios especiales del sistema operativo; como máximo, el sistema de gestión de ficheros completo se considera parte del sistema operativo. Por tanto, es apropiado considerar los elementos básicos de la gestión de ficheros en este libro.

Se comenzará con un primer análisis, pasando a continuación a mostrar varios esquemas de organización de ficheros. Aunque la organización de ficheros está generalmente fuera del ámbito del sistema operativo, es esencial tener una comprensión general de las alternativas comunes para apreciar algunos de los compromisos de diseño relacionados con la gestión de ficheros. El resto del capítulo describe otros temas de la gestión de ficheros.

◆ ◆

12.1. DESCRIPCIÓN BÁSICA

FICHEROS Y SISTEMAS DE FICHEROS

Desde el punto de vista del usuario, una de las partes más importantes de un sistema operativo es el sistema de ficheros. El sistema de ficheros proporciona las abstracciones de recursos típicamente asociadas con el almacenamiento secundario. El sistema de ficheros permite a los usuarios crear colecciones de datos, llamadas ficheros, con propiedades deseables, tales como las siguientes:

- **Existencia a largo plazo.** Los ficheros se almacenan en disco u otro almacenamiento secundario y no desaparece cuando un usuario se desconecta.
- **Compartible entre procesos.** Los ficheros tienen nombres y pueden tener permisos de acceso asociados que permitan controlar la compartición.
- **Estructura.** Dependiendo del sistema de ficheros, un fichero puede tener una estructura interna que es conveniente para aplicaciones particulares. Adicionalmente, los ficheros se pueden organizar en estructuras jerárquicas o más complejas para reflejar las relaciones entre los mismos.

Cualquier sistema de ficheros proporciona no sólo una manera de almacenar los datos organizados como ficheros, sino una colección de funciones que se pueden llevar a cabo sobre ficheros. Algunas operaciones típicas son las siguientes:

- **Crear.** Se define un nuevo fichero y se posiciona dentro de la estructura de ficheros.
- **Borrar.** Se elimina un fichero de la estructura de ficheros y se destruye.
- **Abrir.** Un fichero existente se declara «abierto» por un proceso, permitiendo al proceso realizar funciones sobre dicho fichero.
- **Cerrar.** Un determinado proceso cierra un fichero, de forma que no puede volver a realizar determinadas funciones sobre el mismo, a no ser que vuelva a abrirlo.

- **Leer.** Un proceso lee de un fichero todos los datos o una porción de ellos.
- **Escribir.** Un proceso actualiza un fichero, bien añadiendo nuevos datos que expanden el tamaño del fichero, bien cambiando los valores de elementos de datos existentes en el fichero.

Típicamente, un sistema de ficheros mantiene un conjunto de atributos asociados al fichero. Estos incluyen el propietario, tiempo de creación, tiempo de última modificación, privilegios de acceso, etc.

ESTRUCTURA DE UN FICHERO

Cuatro términos aparecen normalmente cuando se habla sobre ficheros:

- Campo.
- Registro.
- Fichero.
- Base de datos.

Un **campo** es el elemento básico de los datos. Un campo individual contiene un único valor, tal como el apellido de un empleado, una fecha o el valor de un sensor. Se caracteriza por su longitud y el tipo de datos (por ejemplo, ASCII, cadena de caracteres, decimal). Dependiendo del diseño del fichero, el campo puede tener una longitud fija o variable. En este último caso, el campo está formado normalmente por dos o tres subcampos: el valor real almacenado, el nombre del campo, y en algunos casos, la longitud del campo. En otros casos de campos de longitud variable, la longitud del campo se indica mediante el uso de símbolos de demarcación especiales entre campos.

Un **registro** es una colección de campos relacionados que pueden tratarse como una unidad por alguna aplicación. Por ejemplo, un registro de empleado podría contener campos tales como nombre, número de seguridad social, clasificación de trabajo, fecha de contratación, etc. De nuevo, dependiendo del diseño, los registros pueden ser de longitud fija o variable. Un registro tendrá una longitud variable si alguno de sus campos tiene longitud variable o si el número de campos puede variar. En este último caso, cada campo se acompaña normalmente de un nombre de campo. En cualquier caso, el registro completo incluye normalmente un campo longitud.

Un **fichero** es una colección de campos similares. El fichero se trata como una entidad única por parte de los usuarios y las aplicaciones. Los ficheros se pueden referenciar por nombre. Dichos ficheros se pueden crear y borrar. Las restricciones de control de acceso normalmente se aplican a nivel del fichero. Es decir, en un sistema compartido, el acceso a los ficheros completos es permitido o denegado a los usuarios y los programas. En algunos sistemas más sofisticados, tales controles se realizan a nivel de registro o incluso a nivel de campo.

Una **base de datos** es una colección de datos relacionados. Los aspectos esenciales de una base de datos son que la relación que exista entre los elementos de datos sea explícita y que la base de datos se diseña para su uso por parte de varias aplicaciones diferentes. Una base de datos podría contener toda la información relacionada con una organización o proyecto, tal como información de negocio o de estudio científico. La base de datos está formada por uno o más tipos de ficheros. Normalmente, hay un sistema de gestión de base de datos separado del sistema operativo, aunque hace uso de algunos programas de gestión de ficheros.

Los usuarios y las aplicaciones desean utilizar ficheros. Las operaciones típicas que deben soportarse incluyen:

- **Obtener_Todos.** Obtener todos los registros de un fichero. Esta operación se requerirá por aplicaciones que deban procesar toda la información de un fichero de una vez. Por ejemplo, una aplicación que produzca un resumen de la información existente en un fichero necesitaría obtener todos sus registros. Esta operación se asocia frecuentemente con el término *procesamiento secuencial*, porque todos los registros se acceden en secuencia.
- **Obtener_Uno.** Esta operación solicita un único registro. Las aplicaciones interactivas y orientadas a transacciones necesitan esta operación.
- **Obtener_Siguiente.** Esta operación solicita el «siguiente» registro, en alguna secuencia lógica con respecto al registro más recientemente leído. Algunas aplicaciones interactivas, tales como el rellenado de formularios, podrían requerir este tipo de operaciones. Un programa que realiza una búsqueda podría también utilizar este tipo de operación.
- **Obtener_Anterior.** Similar a la operación *Obtener_Siguiente*, pero en este caso el registro al que se accede es el anterior al más recientemente leído.
- **Insertar_Uno.** Insertar un nuevo registro en el fichero. Puede ser necesario que el nuevo registro encaje en una posición específica, para preservar una secuencia del fichero.
- **Borrar_Uno.** Borrar un registro existente. Ciertos enlaces u otras estructuras de datos podrían tener que actualizarse para preservar la secuencia del fichero.
- **Actualizar_Uno.** Obtener un registro, actualizar uno o más de sus campos, y reescribir el registro actualizado en el fichero. De nuevo, puede ser necesario preservar la secuenciación con esta operación. Si la longitud del registro ha cambiado, la operación de actualización es generalmente más difícil que si se preserva la longitud.
- **Obtener_Varios.** Obtener varios registros. Por ejemplo, una aplicación o usuario podría desear obtener todos los registros que satisfacen un cierto conjunto de criterios.

La naturaleza de las operaciones que se realizan más habitualmente sobre ficheros influirá en la forma en que se organiza un fichero, como se discutirá en la Sección 12.2.

Debería destacarse que no todos los sistemas de ficheros utilizan el conjunto de estructuras discutido en esta subsección. En UNIX y sistemas similares, la estructura de fichero básica es sólo un flujo de caracteres. Por ejemplo, un programa C se almacena como un fichero pero no tiene campos físicos, registros u otras estructuras.

SISTEMAS DE GESTIÓN DE FICHEROS

Un sistema de gestión de ficheros es aquel conjunto de software de sistema que proporciona servicios a los usuarios y aplicaciones en el uso de ficheros. Típicamente, la única forma en la que un usuario o aplicación puede acceder a los ficheros es a través del sistema de gestión de ficheros. Esto elimina la necesidad de que el usuario o programador desarrolle software de propósito especial para cada aplicación. Además, proporciona al sistema una forma consistente y bien definida de controlar su recurso más importante. [GROS86] sugiere los siguientes objetivos de un sistema de gestión de ficheros:

- Satisfacer las necesidades de gestión de datos y requisitos del usuario, lo que incluye el almacenamiento de datos y la capacidad de llevar a cabo las operaciones anteriormente mencionadas.
- Garantizar, hasta donde sea posible, que los datos del fichero son válidos.
- Optimizar el rendimiento, desde el punto de vista del sistema en términos de productividad y desde el punto de vista del usuario en términos de tiempo de respuesta.

- Proporcionar soporte de E/S a una variedad de tipos de dispositivos de almacenamiento.
- Minimizar o eliminar la potencial pérdida de datos.
- Proporcionar un conjunto estándar de rutinas de interfaces de E/S a los procesos.
- Proporcionar soporte de E/S a múltiples usuarios, en el caso de sistemas multiusuarios.

Respecto al primer punto, satisfacer los requisitos del usuario depende de una variedad de aplicaciones y del entorno en el cual el sistema de computación se utilizará. Para un sistema interactivo, de propósito general, las siguientes características constituyen un conjunto mínimo de requisitos:

1. Cada usuario debería poder crear, borrar, leer, escribir y modificar ficheros.
2. Cada usuario tendría acceso controlado a los ficheros de otros usuarios.
3. Cada usuario podría controlar qué tipos de accesos se permiten a los ficheros de los usuarios.
4. Cada usuario debe poder reestructurar los ficheros de los usuarios en una forma apropiada al problema.
5. Cada usuario debe ser capaz de mover datos entre ficheros.
6. Cada usuario debe poder copiar y recuperar los ficheros de usuario en caso de daño.
7. Cada usuario debe poder acceder a los ficheros utilizando nombres simbólicos.

Estos objetivos y requisitos deben tenerse en mente cuando se discutan los sistemas de gestión de ficheros.

Arquitectura de un sistema de ficheros

Una forma de conocer el ámbito de la gestión de ficheros es analizar la organización de software típica, como sugiere la Figura 12.1. Por supuesto, distintos sistemas se organizarán de forma diferente, pero esta organización es razonablemente representativa. En el nivel más bajo, los **manejadores de dispositivos** se comunican directamente con los dispositivos periféricos o sus controladores o canales. Un controlador de dispositivo es el responsable de iniciar las operaciones de E/S de un dispositivo y procesar la finalización de una petición de E/S. Para las operaciones sobre ficheros, los dispositivos típicos son los discos y las cintas. Los controladores de dispositivos se consideran normalmente parte del sistema operativo.

El siguiente nivel se denomina **sistema de ficheros básico**, o nivel de **E/S físico**. Esta es la interfaz primaria con el entorno fuera del sistema de computación. Trata con bloques de datos que son intercambiados con discos o sistemas de cintas. Por tanto, este nivel se encarga de la colocación de aquellos bloques del dispositivo de almacenamiento secundario y el *buffering* de dichos bloques en memoria principal. No se encarga de interpretar el contenido de los datos o la estructura de los ficheros. El sistema de ficheros básico es frecuentemente considerado parte del sistema operativo.

El **supervisor de E/S básico** se encarga de todas las iniciaciones y finalizaciones de E/S. En este nivel, las estructuras de control tratan con los dispositivos de E/S, la planificación y el estado de los ficheros. El supervisor de E/S básico selecciona el dispositivo en el cual se van a llevar a cabo las operaciones, basándose en el fichero particular seleccionado. También se encarga de la planificación de disco y cinta para optimizar el rendimiento. A este nivel, se asignan los *buffers* de E/S y la memoria secundaria. El supervisor de E/S básico es parte del sistema operativo.

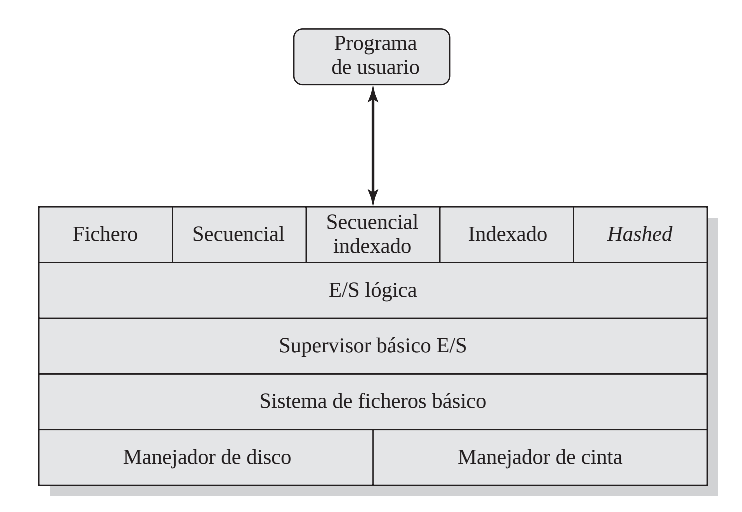


Figura 12.1. Arquitectura software de un sistema de ficheros.

La **E/S lógica** permite a los usuarios y a las aplicaciones acceder a los registros. Por tanto, mientras que el sistema de ficheros básico trata con bloques de datos, el módulo de E/S lógica trata con registros de ficheros. La capa de E/S lógica proporciona una capacidad de E/S de propósito general, a nivel de registros y mantiene datos básicos sobre los ficheros.

El nivel del sistema de ficheros más cercano al usuario es frecuentemente denominado **método de acceso**. Proporciona una interfaz estándar entre las aplicaciones y los sistemas de ficheros y dispositivos que contienen los datos. Diferentes métodos de acceso reflejan diferentes estructuras de ficheros y diferentes formas de acceder y procesar los datos. Algunos de los métodos de acceso más comunes se muestran en la Figura 12.1 y se describen brevemente en la Sección 12.2.

Funciones de gestión de ficheros

Otra forma de ver las funciones de un sistema de ficheros se muestra en la Figura 12.2. Sigamos este diagrama de izquierda a derecha. Los usuarios y programas de aplicaciones interaccionan con el sistema de ficheros por medio de mandatos para crear y borrar ficheros y realizar operaciones sobre los ficheros. Antes de realizar cualquier operación, el sistema de ficheros debe identificar y localizar el fichero seleccionado. Esto requiere el uso de algún tipo de directorio que se utilice para describir la ubicación de todos los ficheros, más sus atributos. Adicionalmente, la mayoría de los sistemas compartidos fuerza el control de acceso de usuario: sólo se permite determinado acceso particular a los usuarios autorizados. Las operaciones básicas que puede realizar un usuario o aplicación se realizan a nivel de registro. El usuario o aplicación ve el fichero como una estructura que organiza los registros, tales como una estructura secuencial (por ejemplo, los registros personales se almacenan alfabéticamente por el apellido). Por tanto, para traducir los mandatos de usuario en mandatos de manipulación de ficheros específicos, debe emplearse el método de acceso apropiado para esta estructura de ficheros.

Mientras que los usuarios y las aplicaciones se preocupan por los registros, la E/S se realiza a nivel de bloque. Por tanto, los registros de un fichero se deben convertir en bloque en la salida y volver a convertir en estructura de registro después de la entrada. Para dar soporte de bloque de E/S, se necesitan varias funciones. Se debe gestionar el almacenamiento secundario. Esto supone asignar ficheros a bloques libres de almacenamiento secundario así como conocer qué bloques están disponibles para nuevos ficheros y utilizados en ficheros existentes. Adicionalmente, se deben planificar las peticiones

de E/S de bloques individuales; este tema se trató en el Capítulo 11. Tanto la planificación de disco como la asignación de ficheros están relacionadas con la optimización del rendimiento. Como se podría esperar, estas funciones, por tanto, necesitan considerarse simultáneamente. Más aún, la optimización dependerá de la estructura de los ficheros y los patrones de acceso. De acuerdo a esto, desarrollar un sistema de gestión de ficheros óptimo, desde el punto de vista del rendimiento, es una tarea excesivamente complicada.

La Figura 12.2 sugiere una división entre las responsabilidades del sistema de gestión de ficheros considerado como una utilidad del sistema y las responsabilidades del sistema operativo, siendo el punto de intersección el procesamiento de registros. Esta división es arbitraria: se utilizan distintas técnicas en diversos sistemas.

El resto de este capítulo describe algunos de los aspectos de diseño sugeridos en la Figura 12.2. Se comienza con una discusión de las organizaciones de los ficheros y los métodos de acceso. Aunque este tema está fuera del ámbito de lo que se considera responsabilidad del sistema operativo, es imposible discutir aspectos de diseño sin una apreciación de la organización y el acceso a los ficheros. Por tanto, se describe el concepto de directorio. El sistema operativo, en nombre del sistema de gestión de ficheros, gestiona frecuentemente los directorios. Los temas restantes tratan sobre los aspectos físicos de E/S de gestión de ficheros y son tratados apropiadamente como aspectos de diseño de sistemas operativos. Uno de estos temas es la forma en la cual los registros lógicos se organizan en bloques físicos. Finalmente, hay temas relacionados con la asignación de ficheros en almacenamiento secundario y la gestión de almacenamiento secundario libre.

12.2. ORGANIZACIÓN Y ACCESO A LOS FICHEROS

En esta sección, se utiliza el término *organización de fichero* para referirse a la estructura lógica de los registros determinados por la forma en la que se acceden. La organización física del fichero en al-

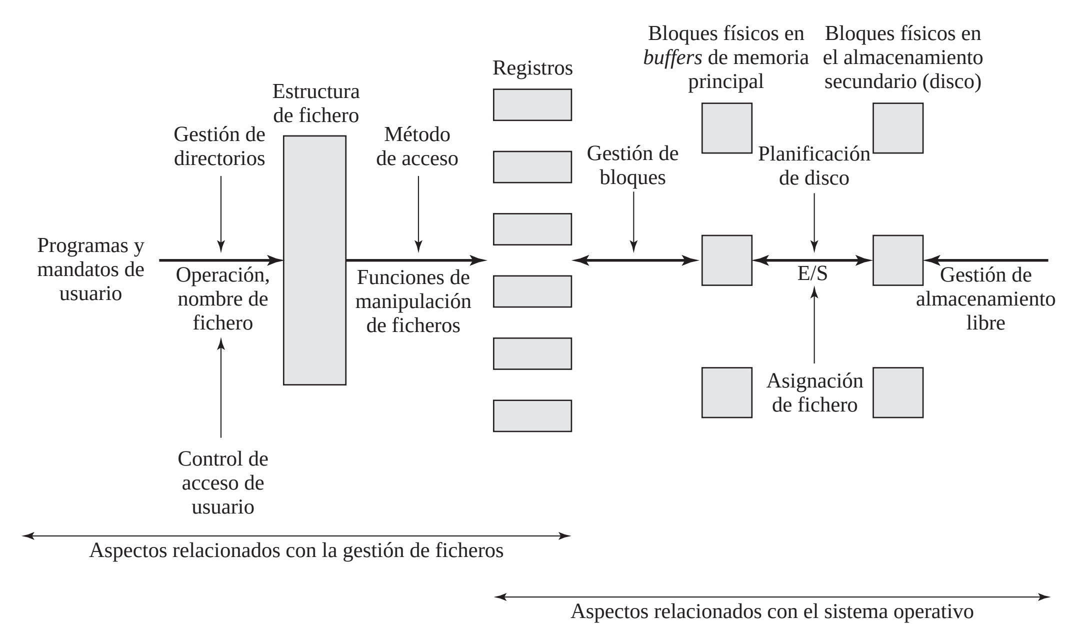


Figura 12.2. Ciclo de instrucción básico.

macenamiento secundario depende de la estrategia de bloques y de asignación de ficheros, temas tratados posteriormente en este capítulo.

Para escoger una organización de ficheros, son importantes varios criterios:

- Tiempo de acceso corto.
- Facilidad de actualización.
- Economía de almacenamiento.
- Mantenimiento sencillo.
- Fiabilidad.

La prioridad relativa de estos criterios dependerá de las aplicaciones que utilizarán el fichero. Por ejemplo, si un fichero se va a procesar sólo en lotes, con acceso a todos los registros cada vez, entonces el acceso rápido a un único registro no es un requisito. Un fichero almacenado en CD-ROM nunca se actualizará, y por tanto la facilidad de actualización no es un aspecto a tener en cuenta en este caso.

Estos criterios pueden entrar en conflicto. Por ejemplo, para facilitar la economía de almacenamiento, debería haber mínima redundancia en los datos. Por otro lado, la redundancia es una medida primaria para incrementar la velocidad de acceso a los datos. Un ejemplo lo constituye el uso de índices.

El número de organizaciones de ficheros alternativas que se han implementado o simplemente propuesto es inmanejablemente largo, incluso para un libro dedicado a los sistemas de ficheros. En este breve resumen, se describen cinco organizaciones fundamentales. La mayoría de las estructuras utilizadas en los sistemas reales cae dentro de una de estas categorías y se puede implementar con una combinación de estas organizaciones. Las cinco organizaciones, de las cuales las cuatro primeras se muestran en la Figura 12.3, son:

- La pila.
- El fichero secuencial.
- El fichero secuencial indexado.
- El fichero indexado.
- El fichero de acceso directo o *hash*.

La Tabla 12.1 resume los aspectos de rendimiento relativo de estas cinco organizaciones[1].

LA PILA

La forma menos complicada de organización de ficheros se puede denominar *pila*. Los datos se almacenan en el orden en el que llegan. Cada registro está formado por un conjunto de datos. El propósito de la pila es simplemente acumular la masa de datos y guardarlos. Los registros podrían tener diferentes campos o similares campos en diferentes órdenes. Por tanto, cada campo debe ser autodescrip-

---

[1] La tabla emplea la notación «O mayúscula», utilizada para caracterizar la complejidad de tiempo de los algoritmos. Una explicación de esta notación se encuentra en un documento en el sitio web de este libro.

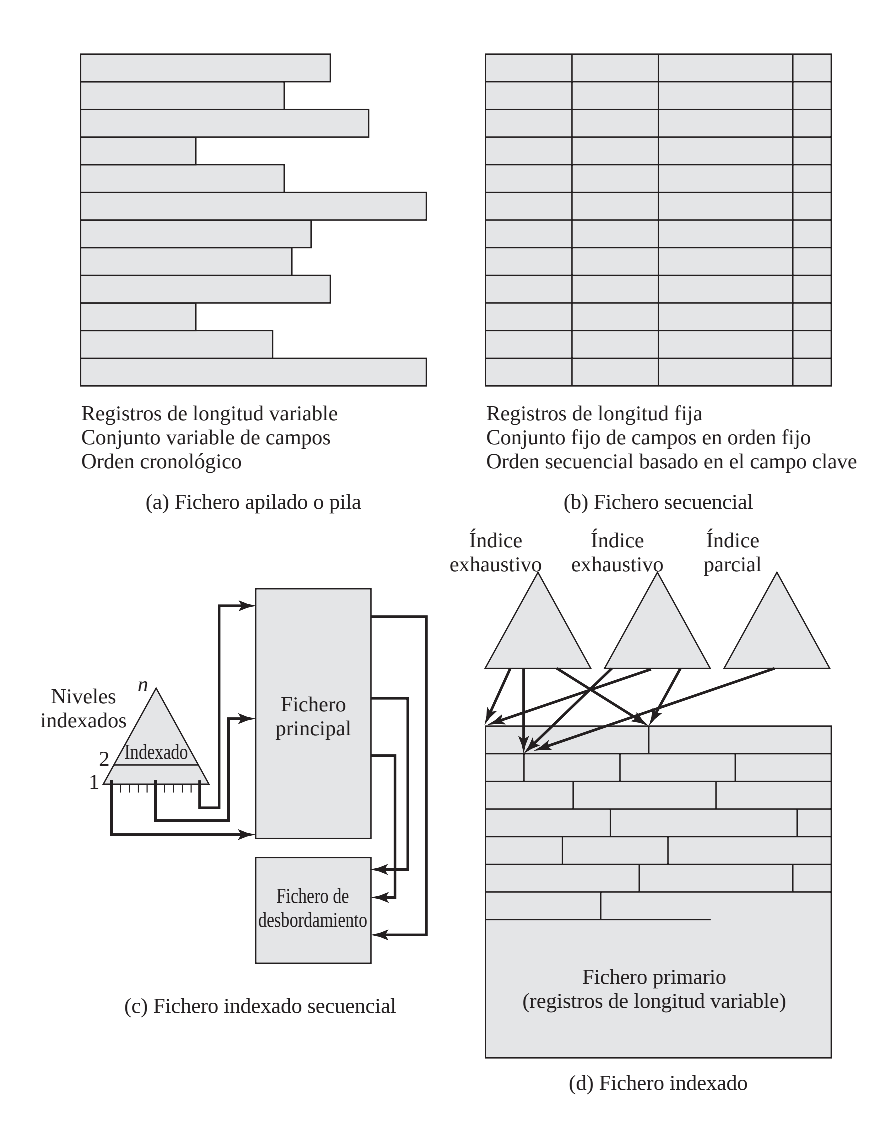


Figura 12.3. Organizaciones comunes de ficheros.

tivo, incluyendo el nombre del campo y el valor. La longitud de cada campo debe ser implícitamente indicada por delimitadores, explícitamente incluida como un subcampo o conocida por defecto para cada tipo de campo.

Dado que no hay estructura en este tipo de fichero, el acceso a los registros se hace mediante búsqueda exhaustiva. Es decir, si se desea encontrar un registro que contiene un campo particular con un valor particular, es necesario examinar cada registro en la pila hasta encontrar el registro deseado o terminar de recorrer todo el fichero. Si se desea encontrar todos los registros que contienen un campo particular o contienen dicho campo con un valor particular, entonces se debe recorrer el fichero entero.

Los ficheros pila se utilizan cuando los datos se recogen y almacenan antes del procesamiento o cuando los datos no son fáciles de organizar. Este tipo de ficheros utiliza el espacio adecuadamente cuando los datos varían en tamaño y estructura, es perfectamente adecuado para búsquedas exhausti-

Tabla 12.1. Grados de rendimiento de las cinco organizaciones de ficheros básicas [WIED87].

Método de fichero	Espacio		Actualización		Adquisición		
	Atributos		Tamaño de registro		Único registro	Sub-conjunto	Exhaustivo
	Variable	Fijo	Igual	Mayor			
Pila	A	B	A	E	E	D	B
Secuencial	F	A	D	F	F	D	A
Indexado secuencial	F	B	B	D	B	D	B
Indexado	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A = Excelente, muy adecuado para este propósito ≈ O(r)
B = Bueno ≈ O(o x r)
C = Adecuado ≈ O(r log n)
D = Requiere algún esfuerzo extra ≈ O(n)
E = Posible con esfuerzo extremo ≈ O(r x n)
F = No razonable para este propósito ≈ $O(n^{>1})$

donde

r = tamaño de los resultados
o = número de registros que desbordan
n = número de registros del fichero

vas y es fácil de actualizar. Sin embargo, más allá de estos usos limitados, este tipo de fichero es inadecuado para la mayoría de las aplicaciones.

EL FICHERO SECUENCIAL

La forma más común de estructura de fichero es el fichero secuencial. En este tipo de ficheros, se utiliza un formato fijo para los registros. Todos los registros son de igual tamaño y están compuestos por el mismo número de campos de longitud fija en un orden específico. Debido a que la longitud y la posición de cada campo son conocidas, sólo se necesita almacenar los valores de los campos; el nombre y longitud de cada campo son atributos de la estructura del fichero.

Un campo particular, normalmente el primer campo de cada registro, se denomina **campo clave**. El campo clave identifica de forma única el registro; por tanto, los valores de la clave de diferentes registros son siempre diferentes. Más aún, los registros se almacenan en secuencia según la clave: orden alfabético para una clave de texto y orden numérico para una clave numérica.

Los ficheros secuenciales se utilizan normalmente en aplicaciones en lotes y son generalmente óptimos para dichas aplicaciones cuando implican el procesamiento de todos los registros (por ejemplo, una aplicación bancaria o de nóminas). La organización de fichero secuencial es el único que se almacena fácilmente en cinta y en disco.

Para aplicaciones interactivas que suponen consultas y/o actualizaciones de registros individuales, el fichero secuencial proporciona un pobre rendimiento. Los accesos requieren una búsqueda en

el fichero para encontrar una clave. Si el fichero entero, o una porción grande del fichero, se pueden llevar a memoria principal simultáneamente, se pueden utilizar técnicas de búsqueda más eficientes. No obstante, acceder a un registro en un fichero secuencial grande implica un procesamiento y retrasos considerables. Las adiciones a los ficheros también presentan problemas. Típicamente, un fichero secuencial se almacena en orden secuencial simple de los registros dentro de los bloques. Es decir, la organización física del fichero en cinta o discos encaja directamente con la organización lógica del fichero. En este caso, el procedimiento normal es colocar nuevos registros en un fichero de pila separado, denominado fichero registro o fichero de transacciones. Periódicamente, un sistema de actualización lleva a cabo la mezcla entre el registro y el fichero maestro para producir un nuevo fichero en la secuencia de claves correcta.

Una alternativa es organizar el fichero secuencial físicamente como una lista enlazada. Uno o más registros se almacenan en cada bloque físico. Cada bloque del disco contiene un puntero al siguiente bloque. La inserción de nuevos registros implica una manipulación de punteros, pero no requiere que los nuevos registros ocupen una posición de bloque físico específica. Por tanto, se obtienen ventajas a cambio de incrementar el procesamiento y la sobrecarga.

EL FICHERO SECUENCIAL INDEXADO

Una técnica popular para eliminar las desventajas del fichero secuencial es utilizar los ficheros secuenciales indexados. El fichero secuencial indexado mantiene las características clave del fichero secuencial: los registros se organizan en secuencia, basándose en un campo clave. Dos características se añaden: un índice al fichero que da soporte al acceso aleatorio y un fichero de desbordamiento. El índice proporciona una capacidad de búsqueda para alcanzar rápidamente la vecindad de un registro deseado. El fichero de desbordamiento es similar al fichero registro, utilizado con un fichero secuencial, pero se integra de tal forma que un registro en el fichero de desbordamiento se localiza mediante un puntero desde su registro predecesor.

En la estructura secuencial indexada más sencilla, se utiliza un único nivel de indexación. El índice en este caso es un fichero secuencial simple. Cada registro del fichero índice está formado por dos campos: un campo clave, que es el mismo que el campo clave del fichero principal y un puntero al fichero principal. Para encontrar un campo específico, se busca el índice que contenga el mayor valor clave que sea igual o preceda al valor de clave deseado. La búsqueda continúa en el fichero principal en la ubicación indicada por el puntero.

Para comprobar la efectividad de esta técnica, considere un fichero secuencial con 1 millón de registros. Buscar un valor clave particular requerirá medio millón de accesos a los registros de media. Supóngase que se construye un índice que contiene 1000 entradas, con las claves del índice más o menos distribuidas uniformemente en el fichero principal. Encontrar un registro llevará una media de 500 accesos al fichero índice seguido por 500 accesos al fichero principal. La longitud de búsqueda media se reduce de 500.000 a 1000.

Las adiciones al fichero se gestionan de la siguiente forma: cada registro del fichero principal contiene un campo adicional no visible a la aplicación, que es un puntero al fichero de desbordamiento. Cuando se inserta un nuevo registro en el fichero, se añade al fichero de desbordamiento. Se actualiza el registro del fichero principal que inmediatamente precede al nuevo registro en secuencia lógica para contener un puntero al nuevo registro del fichero de desbordamiento. Si el registro inmediatamente precedente está a su vez en el fichero de desbordamiento, entonces se actualiza el puntero de dicho registro. Al igual que el fichero secuencial, el fichero secuencial indexado es ocasionalmente mezclado con el fichero de desbordamiento en modo *batch*.

El fichero secuencial indexado reduce enormemente el tiempo requerido para acceder a un único registro, sin sacrificar la naturaleza secuencial del fichero. Para procesar el fichero entero secuencial-

mente, los registros del fichero principal se procesan en secuencia hasta que se encuentra un puntero al fichero de desbordamiento; a continuación, se accede de forma continua en el fichero de desbordamiento hasta que se encuentra un puntero nulo, momento en el cual se continúa accediendo al fichero principal desde el lugar en que se dejó.

Para proporcionar incluso mayor eficiencia en el acceso, se pueden utilizar múltiples niveles de indexación. A continuación, el menor nivel de indexación se trata como un fichero secuencial y se crea un fichero índice de mayor nivel para dicho fichero. Se construye un índice de bajo nivel con 10.000 entradas. Entonces se puede construir un índice de mayor nivel por cada 100 entradas. La búsqueda comienza en el índice de mayor nivel (longitud media = 50 accesos) para encontrar un punto de entrada en el índice de menor nivel. Por tanto, este índice se procesa (longitud media = 50) para encontrar un punto de entrada en el fichero principal, en el cual se busca de nuevo. Por tanto, la longitud media de búsqueda se ha reducido de 500.000 a 1000, y de 1000 a 150.

EL FICHERO INDEXADO

El fichero secuencial indexado elimina una de las limitaciones del fichero secuencial: el procesamiento efectivo se limita a las búsquedas que se basan en un único campo del fichero. Cuando es necesario buscar por algún otro atributo que no sea el campo clave, ambas formas de ficheros secuenciales son inadecuadas. En algunas aplicaciones, esta flexibilidad es deseable.

Para lograr esta flexibilidad, se necesita una estructura que emplea múltiples índices, uno por cada tipo de campo que puede estar sujeto a una búsqueda. En el fichero indexado general, se abandonan los conceptos de secuencialidad y clave única. Los registros se acceden sólo a través de sus índices. El resultado es que no hay restricción en la colocación de los registros siempre que al menos un puntero en un índice se refiera a dicho registro. Además, se pueden emplear registros de longitud variable.

Se utilizan dos tipos de índice. Un índice exhaustivo contiene una entrada por cada registro del fichero principal. Para facilitar la búsqueda, el índice a su vez está organizado como un fichero secuencial. Un índice parcial contiene entradas a registros donde el campo de interés existe. Con registros de longitud variable, algunos registros no contendrán todos los campos. Cuando se añade un nuevo registro al fichero principal, todos los ficheros índices deben actualizarse.

Los ficheros índices se utilizan frecuentemente en aplicaciones donde la temporización de la información es crítica y donde los datos casi nunca se procesan exhaustivamente. Ejemplos de este tipo de aplicación son los sistemas de reservas de aerolíneas y los sistemas de control de inventario.

EL FICHERO DE ACCESO DIRECTO O *HASH*

El fichero de acceso directo, o *hash*, explota la capacidad encontrada en los discos para acceder directamente a cualquier bloque de una dirección conocida. Al igual que los ficheros secuenciales y secuenciales indexados, se requiere una clave para cada registro. Sin embargo, en este tipo de ficheros no existe el concepto de ordenación secuencial.

El fichero directo hace uso de una función *hash* sobre un valor clave. Esta función se explicó en el Apéndice 8A. La Figura 8.27b muestra el tipo de organización *hash* con un fichero de desbordamiento que se utiliza normalmente en un fichero *hash*.

Los ficheros directos se utilizan frecuentemente cuando se requiere un acceso muy rápido, los registros son de tamaño fijo y los registros se acceden de uno en uno. Ejemplos de este tipo de estructura son los directorios, las tablas de precios, los inventarios y las listas de nombres.

12.3. DIRECTORIOS

CONTENIDO

Asociado con cualquier sistema de gestión de ficheros y colección de ficheros, se encuentra el concepto de directorio. El directorio contiene información sobre los ficheros, incluyendo atributos, ubicación y propiedad. Gran parte de esta información, especialmente la que concierne a almacenamiento, la gestiona el sistema operativo. El directorio es a su vez un fichero, accesible por varias rutinas de gestión de ficheros. Aunque parte de la información de los directorios está disponible para los usuarios y las aplicaciones, esto se proporciona generalmente de forma indirecta por las rutinas del sistema.

La Tabla 12.2 muestra la información normalmente utilizada en el directorio por cada fichero del sistema. Desde el punto de vista del usuario, el directorio proporciona una proyección entre los nombres de ficheros, conocidos para los usuarios y las aplicaciones, y los ficheros en sí. Por tanto, cada entrada del fichero incluye el nombre del fichero. Prácticamente todos los sistemas tratan con diferentes tipos de ficheros y distintas organizaciones de ficheros, y esta información también se proporciona. Una importante categoría de información sobre cada fichero trata sobre el almacenamiento, incluyendo su ubicación y tamaño. En sistemas compartidos, es también importante proporcionar información que se utilice para controlar el acceso a los ficheros. Típicamente, un usuario es el propietario del fichero y puede conceder ciertos privilegios de acceso a otros usuarios. Finalmente, se utiliza información de uso para gestionar la utilización actual del fichero y registrar la historia de su uso.

ESTRUCTURA

La forma en la que la información de la Tabla 12.2 se almacena difiere ampliamente entre varios sistemas. Parte de la información se puede almacenar en un registro cabecera asociado con el fichero; esto reduce la cantidad de almacenamiento requerido para el directorio, haciendo más fácil almacenar el directorio o parte del directorio en memoria principal, a fin de incrementar la velocidad.

La forma más sencilla de estructura para un directorio es una lista de entradas, una por cada fichero. Esta estructura se podría representar como un fichero secuencial simple, con el nombre del fichero actuando como clave. En algunos sistemas iniciales monousuario, se ha utilizado esta técnica. Sin embargo, esta técnica es inadecuada cuando múltiples usuarios comparten el sistema o cuando un único usuario tiene muchos ficheros.

Para comprender los requisitos de la estructura de un fichero, es útil considerar los tipos de operaciones que se pueden llevar a cabo sobre directorios:

- **Buscar.** Cuando un usuario o aplicación referencia un fichero, el directorio debe permitir encontrar la entrada correspondiente a dicho fichero.
- **Crear fichero.** Cuando se crea un nuevo fichero, se debe añadir una entrada al directorio.
- **Borrar fichero.** Cuando se borra un fichero, se debe eliminar una entrada del directorio.
- **Listar directorio.** Se puede solicitar ver el directorio completo o una porción del mismo. Generalmente, el usuario solicita esta petición y como resultado obtiene un listado de todos los ficheros de los cuales es propietario, más algunos de los atributos de cada fichero (por ejemplo, información de control de acceso, información de uso).
- **Actualizar directorio.** Debido a que algunos atributos se almacenan en el directorio, un cambio en uno de estos atributos requiere un cambio en la entrada de directorio correspondiente.

Tabla 12.2. Elementos de información de un directorio.

	Información básica
Nombre de fichero	Nombre escogido por el creador (usuario o programa). Debe ser único dentro de un directorio específico.
Tipo de fichero	Por ejemplo: texto, binario, módulo de carga, etc.
Organización de fichero	Para sistemas que soportan diferentes organizaciones.
	Información de direccionamiento
Volumen	Indica el dispositivo en el cual se almacena el fichero.
Dirección inicial	Dirección física inicial en almacenamiento secundario (por ejemplo, cilindro, pista y número de bloque en disco).
Tamaño utilizado	Tamaño actual del fichero en bytes, palabras o bloques.
Tamaño asignado	Tamaño máximo del fichero.
	Información de control de acceso
Propietario	Usuario que tiene el control del fichero. El propietario puede conceder/denegar acceso a otros usuarios y cambiar estos privilegios.
Información de acceso	Una versión sencilla de este elemento incluye el nombre de usuario y clave para cada usuario autorizado.
Acciones permitidas	Controla la lectura, escritura, ejecución y la transmisión a través de la red.
	Información de uso
Fecha de creación	Fecha en la que el fichero se coloca por vez primera en el directorio.
Identidad del creador	Normalmente aunque no necesariamente el propietario actual.
Fecha de último acceso de lectura	Fecha de la última vez que se leyó un registro.
Identidad de último lector	Usuario que hizo la última lectura.
Fecha de último acceso de modificación	Fecha de la última actualización, inserción o borrado.
Identidad de último modificador	Usuario que hizo la última modificación.
Fecha de la última copia de seguridad	Fecha de la última vez que el fichero fue copiado en otro medio de almacenamiento.
Uso actual	Información sobre la actividad actual sobre el fichero, tal como proceso o procesos que tienen el fichero abierto, si está bloqueado por un proceso y si el fichero ha sido actualizado en memoria principal pero no en disco.

La lista sencilla no es adecuada para dar soporte a estas operaciones. Considérese las necesidades de un único usuario. Los usuarios podrían tener muchos tipos de ficheros, incluyendo ficheros de procesamiento de texto, gráficos, hojas de cálculo, etc. Al usuario le podría gustar tener estos ficheros organizados por proyecto, por tipo, o de alguna forma conveniente. Si el directorio es una lista secuencial simple, no proporciona ayuda para organizar los ficheros y fuerza a los usuarios a tener cuidado de no utilizar el mismo nombre para dos tipos de ficheros diferentes. El problema se agrava en un sistema compartido. El nombrado único se convierte en un problema serio. Más aún, es difícil conceder porciones del directorio completo a los usuarios cuando no hay una estructura inherente en el directorio.

Una primera solución para resolver estos problemas sería pasar a un esquema de dos niveles. En este caso, hay un directorio por cada usuario y un directorio maestro. El directorio maestro tiene una entrada por cada directorio usuario, proporcionando información sobre dirección y control de acceso. Cada directorio de usuario es una lista simple de los ficheros de dicho usuario. Esto implica que los nombres deben ser únicos sólo dentro de la colección de los ficheros de un único usuario y que el sistema de ficheros puede fácilmente asegurar las restricciones de acceso de los directorios. Sin embargo, aún no proporciona ayuda a los usuarios para estructurar su colección de ficheros.

Una técnica más potente y flexible, que es casi universalmente adoptada, es utilizar una estructura jerárquica en forma de árbol (Figura 12.4). Como en la técnica anterior, hay un directorio maestro, que tiene bajo dicho directorio varios directorios de usuario. Cada uno de estos directorios de usuario, a su vez, podría tener subdirectorios y ficheros como entradas. Esto se cumple para todos los niveles: es decir, en cada nivel, un directorio podría estar formado por subdirectorios y/o ficheros.

Falta describir cómo se organiza cada directorio y subdirectorio. El enfoque más sencillo, por supuesto, es almacenar cada directorio como un fichero secuencial. Si los directorios contuvieran un gran número de entradas, dicha organización llevaría a unos tiempos de búsqueda innecesariamente largos. En dicho caso, es preferible una estructura *hash*.

NOMBRADO

Los usuarios necesitan poder referenciar un fichero mediante un nombre simbólico. Claramente, cada fichero en el sistema debe tener un nombre único a fin de que las referencias al mismo no sean ambiguas. Por otro lado, es inaceptable obligar a que los usuarios proporcionen nombres únicos, especialmente en un sistema compartido.

El uso de un directorio estructurado en forma de árbol minimiza la dificultad de asignar nombres únicos. Cualquier fichero del sistema se puede localizar siguiendo un camino desde el directorio raíz o maestro y bajando por las ramas hasta alcanzar el fichero. El conjunto de nombres de directorios, finalizando en el nombre del fichero, constituye un **nombre de camino** para el fichero. Por ejemplo,

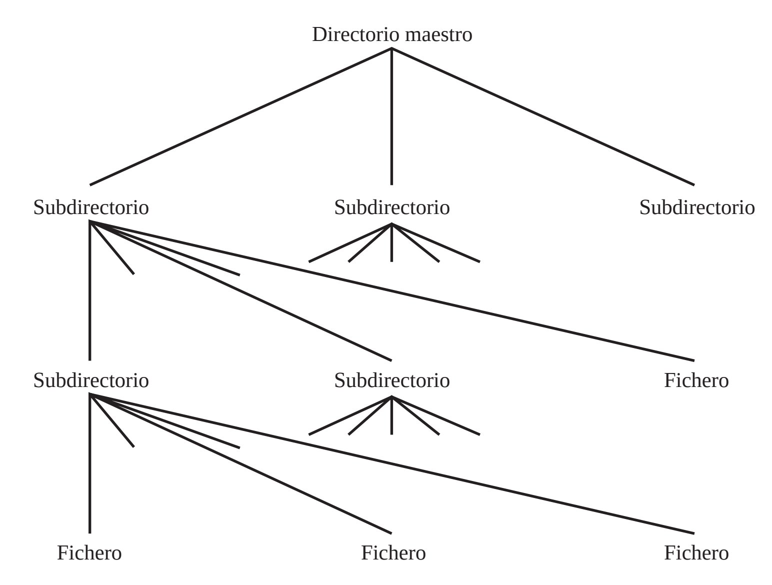


Figura 12.4. Directorio estructurado en forma de árbol.

el fichero de la esquina inferior izquierda de la Figura 12.5 tiene el camino /Usuario_B/Texto/Unidad_A/ABC. La barra se utiliza para delimitar nombres en la secuencia. El nombre del directorio maestro es implícito, porque todos los nombres comienzan en dicho directorio. Obsérvese que es perfectamente aceptable tener varios ficheros con el mismo nombre, siempre que ambos ficheros tengan nombres de camino únicos, lo que equivale a decir que el mismo nombre de fichero se puede utilizar en diferentes directorios. En el ejemplo, hay otro fichero en el sistema con el nombre ABC, pero cuyo nombre completo es /Usuario_B/Dibujo/ABC.

Aunque los nombres de camino facilitan la selección de los nombres de fichero, sería complicado para el usuario tener que escribir el camino completo cada vez que se hace una referencia a un fichero. Normalmente, un usuario interactivo o un proceso está asociado con un directorio actual, que se suele denominar **directorio de trabajo**. Los ficheros se pueden referenciar de forma relativa al direc-

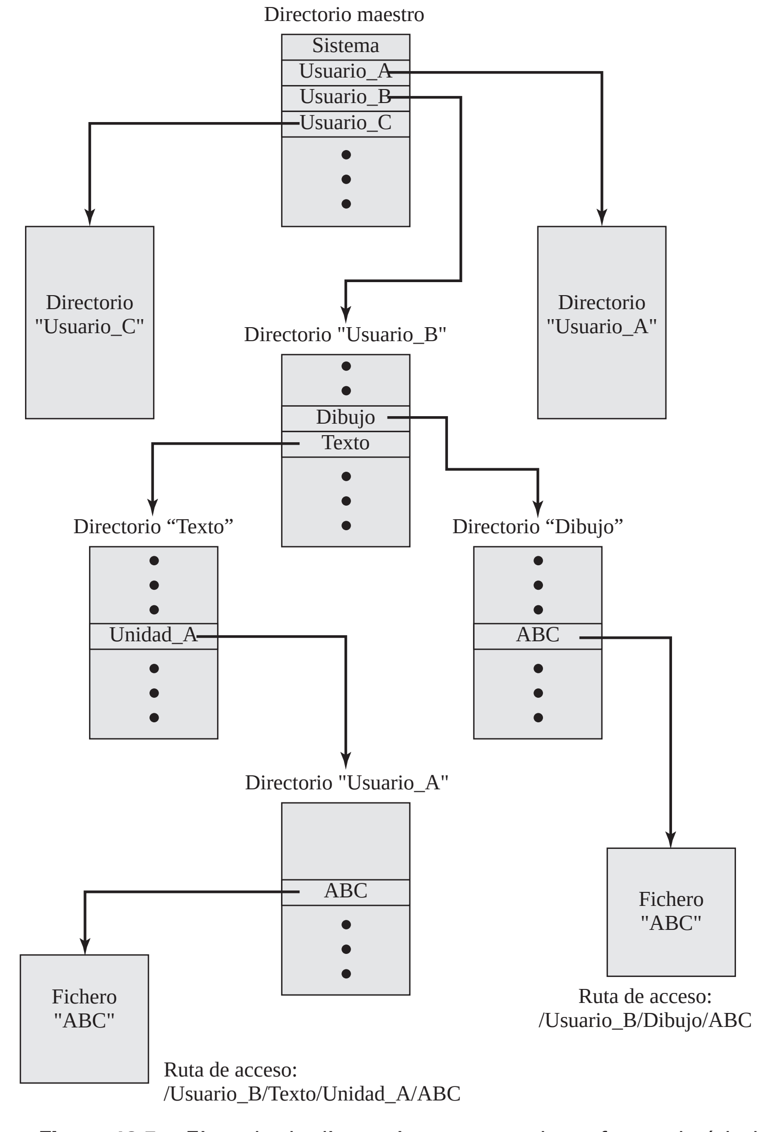


Figura 12.5. Ejemplo de directorio estructurado en forma de árbol.

torio de trabajo. Por ejemplo, si el directorio de trabajo del usuario B es «Texto», entonces el nombre Unidad_A/ABC es suficiente para identificar al fichero de la esquina inferior izquierda de la Figura 12.5. Cuando un usuario interactivo se conecta o cuando se crea un proceso, su directorio de trabajo por omisión es el directorio inicial del usuario. Durante la ejecución, el usuario puede navegar hacia arriba y hacia abajo en el árbol para cambiarse a un directorio de trabajo diferente.

12.4. COMPARTICIÓN DE FICHEROS

En un sistema multiusuario, existe casi siempre el requisito de permitir que los ficheros se compartan entre varios usuarios. Existen dos aspectos relacionados: derechos de acceso y la gestión de acceso simultáneo.

DERECHOS DE ACCESO

El sistema de ficheros debería proporcionar una herramienta flexible para permitir la compartición de ficheros extensiva entre los usuarios. El sistema de ficheros debería proporcionar varias opciones de tal forma que el acceso a un fichero particular se pueda controlar. Típicamente, a los usuarios o grupos de usuarios se les concede ciertos derechos de acceso a un fichero. Se ha utilizado un amplio rango de derechos de acceso. La siguiente lista es representativa de los derechos de acceso que se asignan a un usuario particular para un determinado fichero:

- **Ninguno.** El usuario no puede incluso conocer la existencia del fichero, y por tanto, tampoco puede acceder a él. Para forzar esta restricción, el usuario no tiene permiso de lectura del directorio en el cual se incluye este fichero.
- **Conocimiento.** El usuario puede determinar si el fichero existe y quién es su propietario. El usuario entonces es capaz de solicitar al propietario derechos de acceso adicionales.
- **Ejecución.** El usuario puede cargar y ejecutar un programa pero no copiarlo. Los programas propietarios utilizan normalmente estas restricciones.
- **Lectura.** El usuario puede leer el fichero para cualquier propósito, incluyendo copia y ejecución. Algunos sistemas son capaces de forzar una distinción entre ver y copiar. En el primer caso, el usuario puede ver el contenido del fichero, pero no puede realizar una copia.
- **Adición.** El usuario puede añadir datos al fichero, frecuentemente sólo al final, pero no puede modificar o borrar cualquiera de los contenidos del fichero. Este derecho es útil para recolectar datos de varias fuentes.
- **Actualización.** El usuario puede modificar, borrar o añadir datos al fichero. Esto normalmente incluye escribir el fichero al inicio, reescribirlo completa o parcialmente y borrar todos o una porción de los datos. Algunos sistemas diferencian entre distintos grados de actualización.
- **Cambio de protección.** El usuario puede cambiar los derechos de acceso otorgados a otros usuarios. Normalmente, sólo el propietario del fichero goza de este derecho. En algunos sistemas, el propietario puede extender este derecho a otros. Para prevenir del abuso de este mecanismo, el propietario del fichero normalmente puede especificar qué derechos podría cambiar un usuario con este tipo de permiso.
- **Borrado.** El usuario puede borrar el fichero del sistema de ficheros.

Estos derechos se pueden considerar como una jerarquía, con cada uno de los derechos conteniendo a aquellos que le preceden. Por tanto, si un usuario particular tiene el permiso de actualización

para un fichero concreto, dicho usuario también tendrá los siguientes derechos: conocimiento, ejecución, lectura y adición.

A un usuario se le considera propietario de un determinado fichero, normalmente a la persona que inicialmente creó un fichero. El propietario tiene todos los derechos de acceso listados previamente y puede conceder permisos a otros usuarios. Se pueden proporcionar diferentes accesos a distintas clases de usuarios:

- **Usuario específico.** Usuarios individuales que se designan por el identificador del usuario.
- **Grupos de usuarios.** Un conjunto de usuarios que no se definen individualmente. El sistema debe tener alguna forma de gestionar la membresía de los grupos de usuarios.
- **Todos.** Todos los usuarios que tienen acceso a este sistema. Dichos ficheros se consideran ficheros públicos.

ACCESO SIMULTÁNEO

Cuando se garantiza acceso de adición o actualización de un fichero a más de un usuario, el sistema operativo o sistema de gestión de ficheros debe forzar una disciplina. Una técnica de fuerza bruta consiste en permitir al usuario bloquear el fichero completo cuando se va a actualizar. Un control de grano más fino implica el bloqueo de registros individuales durante la actualización. Esencialmente, este es el problema de los lectores/escritores discutido en el Capítulo 5. Se deben tratar aspectos de exclusión mutua e interbloqueos a la hora de diseñar capacidades de acceso compartidas.

12.5. BLOQUES Y REGISTROS

Como se indica en la Figura 12.2, los registros son las unidades lógicas de acceso de un fichero estructurado, mientras que los bloques son las unidades de E/S con almacenamiento secundario. Para que la E/S se pueda realizar, los registros se deben organizar como bloques[2].

Hay varios aspectos a considerar. Primero, ¿los bloques deberían ser de longitud fija o variable? En la mayoría de los sistemas, los bloques tienen longitud fija. Esto simplifica la E/S, la asignación de *buffers* en memoria principal y la organización de bloques en almacenamiento secundario. A continuación, ¿cuál debería ser el tamaño relativo de un bloque comparado con el tamaño de registro medio? El compromiso es el siguiente: cuanto mayor sea el bloque, más registros se transferirán en una operación de E/S. Si se procesa un fichero de forma secuencial, esto supone una ventaja, porque se reduce el número de operaciones de E/S utilizando bloques mayores, y por tanto, acelerando el procesamiento. Por otro lado, si los registros se acceden de forma aleatoria y no se observa ninguna proximidad de referencias, entonces utilizar bloques más grandes supone transferencias innecesarias de registros no utilizados. Sin embargo, si se combina la frecuencia de operaciones secuenciales con la potencialidad de proximidad de referencias, se puede decir que el tiempo de transferencia de E/S se reduce utilizando bloques mayores. La preocupación viene por el hecho de que bloques más grandes requieren *buffers* de E/S mayores, haciendo la gestión de *buffers* más difícil.

Dado el tamaño de un bloque, se pueden utilizar tres métodos:

---

[2] En contraposición a un fichero que se trata sólo como una ristra de bytes, como en el sistema de ficheros UNIX.

- **Bloques fijos.** Se utilizan registros de longitud fija y se almacenan en un bloque un número integral de registros. Podría haber espacio no utilizado al final de cada bloque. Esto se denomina fragmentación interna.
- **Bloques expandidos de longitud variable.** Se utilizan registros de longitud variable y se empaquetan en bloques sin dejar espacio no utilizado. Por tanto, algunos registros deben expandirse a lo largo de dos bloques, con su continuación indicada por un puntero al bloque sucesor.
- **Bloques no expandidos de longitud variable.** Se utilizan registros de longitud variable, pero no se emplea expansión. Hay espacio malgastado en la mayoría de los bloques debido a la incapacidad para utilizar el resto de un bloque si el siguiente registro es mayor que el espacio no utilizado restante.

La Figura 12.6 ilustra estos métodos asumiendo que los ficheros se almacenan en bloques secuenciales del disco. El efecto no cambiaría si se utilizara algún otro esquema de asignación de ficheros (*véase* Sección 12.6).

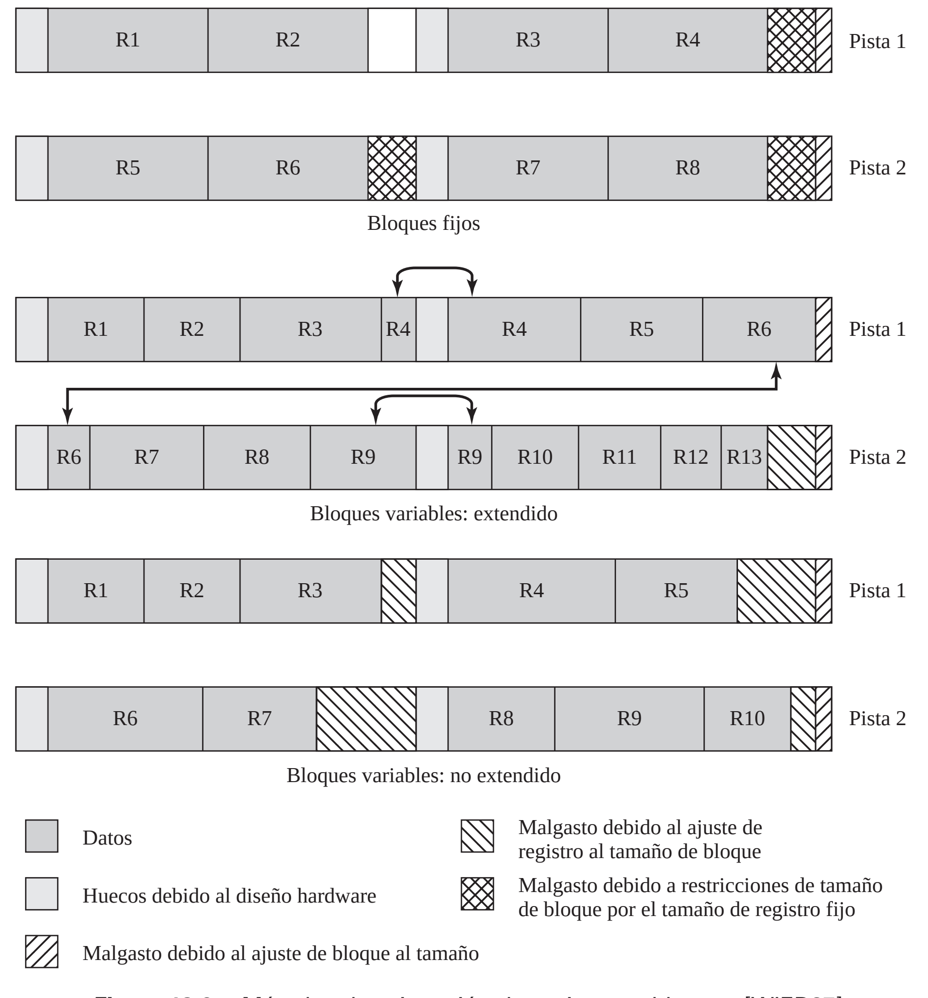


Figura 12.6. Métodos de asignación de registros a bloques [WIED87].

Utilizar bloques fijos es el modo común para ficheros secuenciales con registros de longitud fija. Los bloques expandidos de longitud variable son eficientes respecto al almacenamiento y no limitan el tamaño de los registros. Sin embargo, esta técnica es difícil de implementar. Los registros que expanden dos bloques requieren dos operaciones de E/S y los ficheros son difíciles de actualizar, sin tener en cuenta la organización. Los bloques no expandidos de longitud variable implican espacio malgastado y limitan el tamaño del registro al tamaño de un bloque.

La técnica de bloques y registros utilizada podría interaccionar con el hardware de memoria virtual, si ésta se utilizara. En un entorno de memoria virtual, sería deseable utilizar la página como unidad básica de transferencia. Las páginas son generalmente bastante pequeñas, de forma que es impracticable tratar una página como un bloque para bloques no expandidos. Análogamente, algunos sistemas combinan múltiples páginas para crear un bloque más grande con propósitos de E/S. Esta técnica se utiliza para ficheros VSAM en mainframes IBM.

12.6. GESTIÓN DE ALMACENAMIENTO SECUNDARIO

En almacenamiento secundario, un fichero está compuesto por una colección de bloques. El sistema operativo o sistema de gestión de ficheros es responsable de asignar bloques a los ficheros. Esto supone dos aspectos relacionados con la gestión. Primero, se debe asignar espacio de almacenamiento secundario a los ficheros y segundo, es necesario guardar una traza del espacio disponible para su asignación. Veremos que estas dos tareas están relacionadas; es decir, la técnica seleccionada para asignación de ficheros podría influir en la técnica seleccionada para gestión del espacio libre. Más aún, se verá que existe una interacción entre la estructura de los ficheros y las políticas de asignación.

Se comenzará esta sección analizando alternativas para asignación de ficheros en un único disco. A continuación se describen aspectos relacionados con la gestión de espacio libre y finalmente se discutirá la fiabilidad.

ASIGNACIÓN DE FICHEROS

Varios aspectos están involucrados en la asignación de ficheros:

1. Cuando se crea un fichero nuevo, ¿se asigna de una vez el espacio máximo requerido para el fichero?
2. El espacio se asigna a un fichero como una o más unidades contiguas, lo que se denomina porción. El tamaño de una porción puede ir desde un único bloque al fichero completo. ¿Qué tamaño de porción debería utilizarse para asignación de ficheros?
3. ¿Qué clase de estructura de datos o tabla se utiliza para guardar traza de las porciones asignadas para un fichero? Un ejemplo de dicha estructura es una **tabla de asignación de ficheros** (*File Allocation Table*, FAT), encontrado en DOS y otros sistemas.

Se examinarán estos aspectos a continuación.

Preasignación frente a asignación dinámica

Una política de preasignación requiere que el tamaño máximo de un fichero sea declarado en tiempo de creación de fichero. En varios casos, tales como las compilaciones de programas, la producción de ficheros de datos resumen, o la transferencia de un fichero desde otro sistema sobre la red de comuni-

cación, se puede estimar este valor de forma fiable. Sin embargo, para muchas aplicaciones, es difícil sino imposible, estimar fiablemente el tamaño máximo potencial del fichero. En dichos casos, los usuarios y los programadores de aplicaciones tenderían a sobrestimar el tamaño de fichero de forma que no se queden sin espacio. Esto claramente supone malgasto desde el punto de vista de la asignación de espacio de almacenamiento. Por tanto, hay ventajas en el uso de la gestión dinámica, que asigna espacio a un fichero en porciones cuando se necesite.

Tamaño de porción

El segundo aspecto listado es el tamaño de la porción asignado a un fichero. En un extremo, se puede asignar una porción suficientemente grande para contener el fichero completo. En el otro extremo, para el espacio en el disco se puede asignar un bloque cada vez. Para escoger un tamaño de porción, debe existir un compromiso entre la eficiencia desde el punto de vista de un único fichero y la eficiencia del sistema completo. [WIED87] lista cuatro aspectos a considerar en este compromiso:

1. La contigüidad del espacio incrementa el rendimiento, especialmente para operaciones *Obtener_Siguiente* y para transacciones ejecutándose en un sistema operativo orientado a transacciones.
2. Utilizar un gran número de porciones pequeñas incrementa el tamaño de las tablas necesarias para gestionar la información de asignación.
3. Utilizar porciones de tamaño fijo (por ejemplo, bloques) simplifica la reasignación de espacio.
4. Utilizar porciones de tamaño variable o pequeñas de tamaño fijo minimiza el espacio malgastado debido a la sobreasignación.

Por supuesto, estos elementos interaccionan y se deben considerar conjuntamente. Por tanto, existen dos alternativas principales:

- **Porciones variables, grandes y contiguas.** Esta alternativa proporciona el mejor rendimiento. El tamaño variable evita malgastar espacio, y las tablas de asignación de ficheros son pequeñas. Sin embargo, el espacio es difícil de reutilizar.
- **Bloques.** Pequeñas porciones fijas proporcionan mayor flexibilidad. Podrían requerir grandes tablas o estructuras complejas para su asignación. La contigüidad ha sido abandonada como meta primaria; los bloques se asignan según se necesite.

Cada opción es compatible con preasignación o asignación dinámica. En el caso de porciones variables, grandes y contiguas, al fichero se le preasigna un grupo de bloques contiguos. Esto elimina la necesidad de una tabla de asignación de ficheros; todo lo que se requiere es un puntero al primer bloque y el número de bloques asignados. En el caso de los bloques, todas las porciones requeridas se asignan a la vez. Esto significa que la tabla de asignación de ficheros para el fichero es de tamaño fijo.

Con porciones de tamaño variable, es necesario preocuparse de la fragmentación del espacio libre. Este aspecto se trató cuando se consideró la memoria principal particionada en el Capítulo 7. Las siguientes estrategias son posibles:

- **Primer ajuste.** Escoger el primer grupo contiguo no utilizado de bloques de tamaño suficiente desde una lista de bloques libres.
- **Siguiente ajuste.** Escoger el grupo más pequeño no utilizado que sea de suficiente tamaño.

- **Ajuste más próximo.** Escoger el grupo no utilizado de tamaño suficiente que sea más cercano a la asignación previa para el fichero de manera que se incremente la proximidad.

No está claro qué estrategia es mejor. La dificultad de modelar estrategias alternativas es el hecho de que muchos factores interaccionen, incluyendo tipos de ficheros, patrones de acceso a ficheros, grado de multiprogramación, otros factores de rendimiento del sistema, *caching* de disco, planificación de disco y otros.

Métodos de asignación de ficheros

Habiendo analizado los aspectos de preasignación frente a la asignación dinámica y el tamaño de las porciones, estamos en posición de considerar métodos específicos de asignación de ficheros. Tres métodos son de uso común: contiguo, encadenado e indexado. La Tabla 12.3 resume algunas de las características de cada método.

Tabla 12.3. Métodos de asignación de ficheros.

	Contiguos	Encadenado	Indexado	
¿Preasignación?	Necesaria	Posible	Posible	
¿Porciones de tamaño fijo o variable?	Variable	Bloques fijos	Bloques fijos	Variable
Tamaño de porción	Grande	Pequeño	Pequeño	Medio
Frecuencia de asignación	Una vez	Pequeña a alta	Alta	Baja
Tiempo a asignar	Medio	Largo	Corto	Medio
Tamaño de tabla de asignación de ficheros	Una entrada	Una entrada	Grande	Medio

Con **asignación contigua**, se asigna un único conjunto contiguo de bloques en tiempo de creación de los ficheros (Figura 12.7). Por tanto, hay una estrategia de preasignación que utiliza porciones de tamaño variable. La tabla de asignación de ficheros necesita sólo una entrada para cada fichero, mostrando el bloque inicial y la longitud del fichero. La asignación contigua es la mejor desde el punto de vista del fichero secuencial individual. Múltiples bloques se pueden leer de una vez para mejorar el rendimiento de E/S en procesamiento secuencial. Es también fácil obtener un único bloque. Por ejemplo, si un fichero comienza en el bloque b y se quiere acceder al bloque i-ésimo del fichero, su ubicación en almacenamiento secundario es simplemente $b + i - 1$. La asignación contigua presenta algunos problemas. Existirá fragmentación externa, haciendo difícil encontrar bloques contiguos de espacio de suficiente longitud. De vez en cuando, será necesario llevar a cabo un algoritmo de compactación para liberar espacio adicional en el disco (Figura 12.8). Además, con preasignación, es necesario declarar el tamaño del fichero en el tiempo de creación, con los problemas mencionados anteriormente.

En el extremo opuesto de la asignación contigua está la **asignación encadenada** (Figura 12.9). Típicamente, la asignación se realiza a nivel de bloques individuales. Cada bloque contiene un puntero al siguiente bloque en la cadena. De nuevo, la tabla de asignación de ficheros necesita sólo una entrada para cada fichero, mostrando el bloque inicial y la longitud del fichero. Aunque

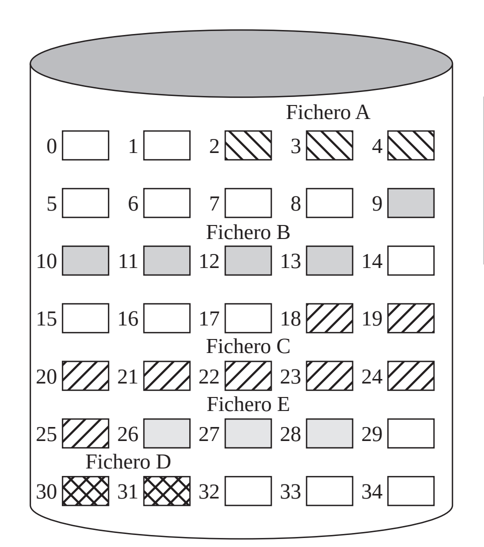


Tabla de asignación de ficheros

Nombre de fichero	Bloque inicial	Longitud
Fichero A	2	3
Fichero B	9	5
Fichero C	18	8
Fichero D	30	2
Fichero E	26	3

Figura 12.7. Asignación de fichero contiguo.

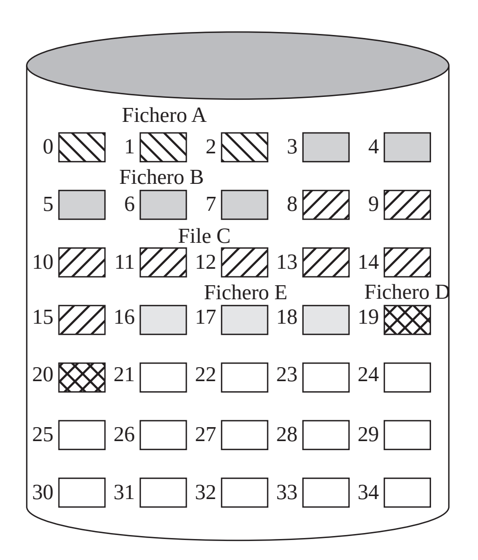


Tabla de asignación de ficheros

Nombre de fichero	Bloque inicial	Longitud
Fichero A	0	3
Fichero B	3	5
Fichero C	8	8
Fichero D	19	2
Fichero E	16	3

Figura 12.8. Asignación de fichero contiguo (después de la compactación).

la preasignación es posible, es más común asignar bloques cuando se necesita. La selección de bloques es ahora una cuestión sencilla: cualquier bloque libre se puede añadir a una cadena. No hay fragmentación externa de la que preocuparse porque sólo se necesita un bloque cada vez. Este tipo de organización física se adapta mejor a ficheros secuenciales que se procesan secuencialmente. Seleccionar un bloque individual de un fichero requiere seguir la cadena hasta alcanzar el bloque deseado.

Una consecuencia del encadenamiento, tal como se describe, es que no existe principio de proximidad. Por tanto, si es necesario traer varios bloques de fichero a la vez, como en el procesamiento secuencial, se requiere una serie de accesos a diferentes partes del disco. Esto es tal vez un efecto más

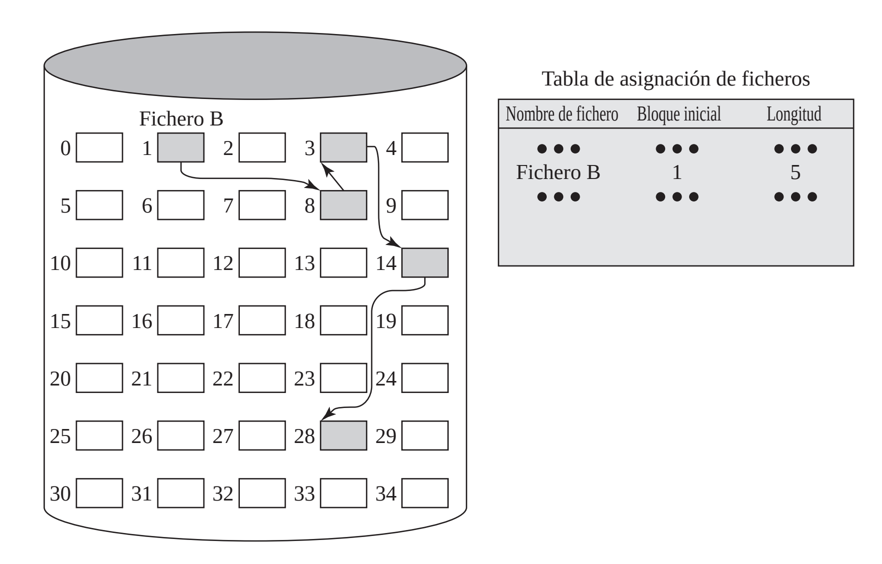


Figura 12.9. Asignación encadenada.

significativo en un sistema monousuario, pero también podría ser preocupante en el caso de un sistema compartido. Para resolver este problema, algunos sistemas consolidan ficheros periódicamente (Figura 12.10).

La **asignación indexada** resuelve muchos de los problemas de la asignación contigua y encadenada. En este caso, la tabla de asignación de ficheros contiene un índice separado de un nivel por cada fichero; el índice tiene una entrada por cada porción asignada al fichero. Típicamente, los índices de fichero no se almacenan físicamente como parte de la tabla de asignación de ficheros. Por el contrario, el índice de ficheros para un fichero se guarda en un bloque separado y la entrada para fichero en la tabla de asignación de ficheros apunta a dicho bloque. La asignación puede realizarse me-

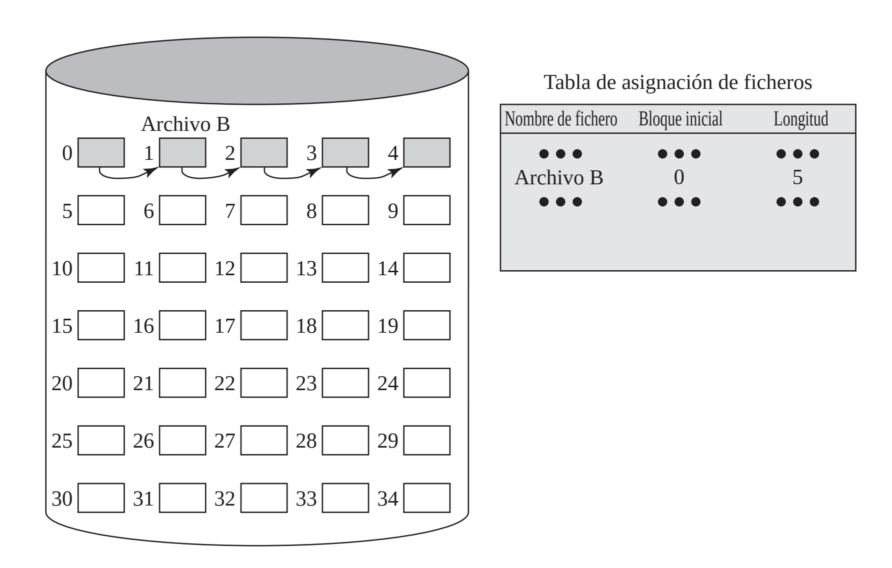


Figura 12.10. Asignación encadenada (después de la consolidación).

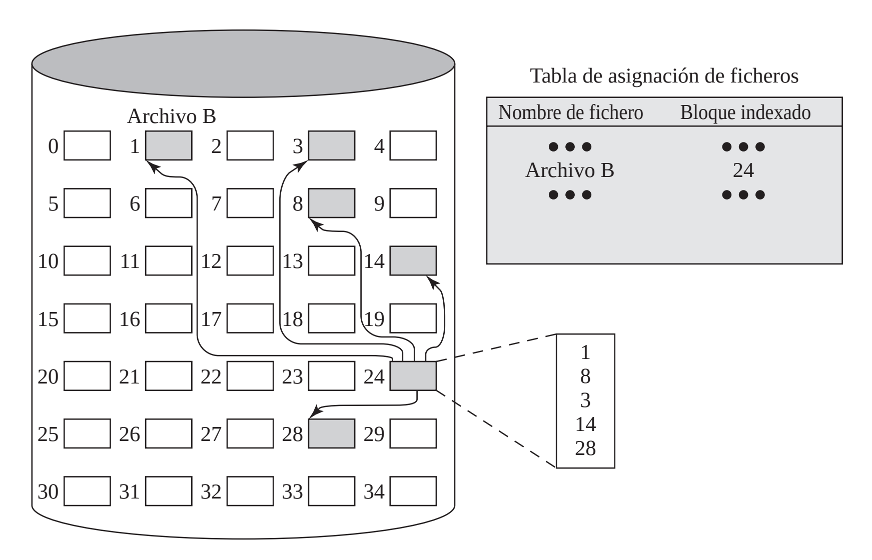


Figura 12.11. Asignación indexada con porciones de bloques.

diante bloques de tamaño fijo (Figura 12.11) o porciones de tamaño variable (Figura 12.12). La asignación por bloques elimina la fragmentación externa, mientras que la asignación por porciones de tamaño variable mejora la proximidad. En cualquier caso, la consolidación de ficheros se puede realizar de vez en cuando. La consolidación de ficheros reduce el tamaño del índice en el caso de porciones de tamaño variable, pero no en el caso de asignación de bloques. La asignación indexada da soporte tanto a acceso secuencial como directo a los ficheros y por tanto es la forma más popular de asignación de ficheros.

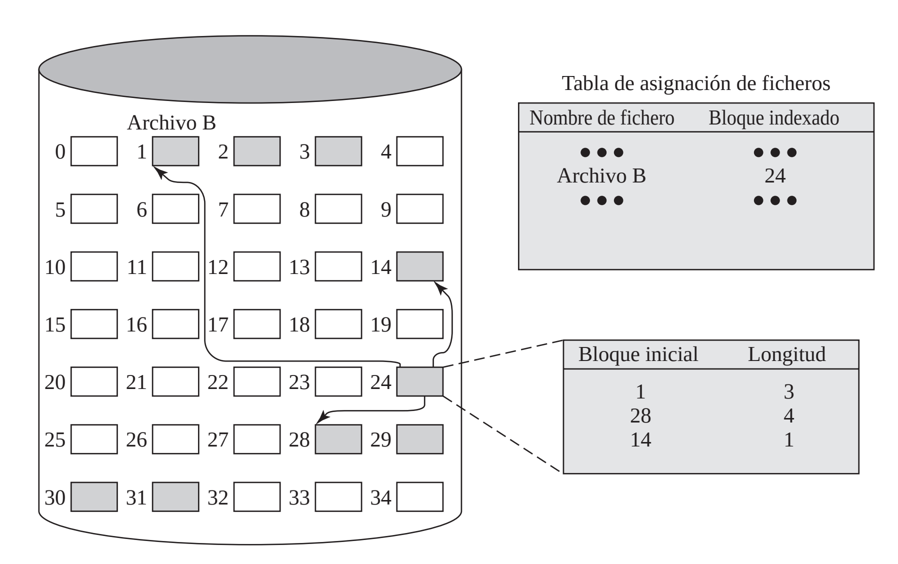


Figura 12.12. Asignación indexada con porciones de tamaño variable.

GESTIÓN DE ESPACIO LIBRE

De la misma forma que se asigna el espacio a los ficheros, así también se debe gestionar el espacio que no está actualmente asignado a ningún fichero. Para llevar a cabo cualquiera de las técnicas de asignación de ficheros descritas previamente, es necesario saber qué bloques están disponibles en el disco. Por tanto se necesita una **tabla de asignación de disco** en adición a la tabla de asignación de ficheros. Se discutirán aquí varias técnicas que se han implementado.

Tablas de bits

Este método utiliza un vector que está formado por un bit por cada bloque en el disco. Cada entrada 0 corresponde a un bloque libre y cada 1 corresponde a un bloque en uso. Por ejemplo, para la composición de disco de la Figura 12.7, se necesita un vector de longitud 35 con el siguiente contenido:

00111000011111000011111111111011000

Una tabla de bits tiene la ventaja de que es relativamente fácil encontrar un bloque libre o un grupo contiguo de bloques libres. Por tanto, una tabla de bits trabaja bien con cualquiera de los métodos de asignación de ficheros descrito. Otra ventaja es que esta estructura es tan pequeña como sea posible. La cantidad de memoria (en bytes) requerida para un mapa de bits de bloques es

$$\frac{\text{Tamaño de disco en bytes}}{8 \times \text{tamaño de bloque del sistema de ficheros}}$$

Por tanto, para un disco de 16 Gbytes con bloques de 512 bits, la tabla de bits ocupa 4 Mbytes. ¿Se puede almacenar la tabla de bits de 4 Mbytes en memoria principal? Si es así, entonces a la tabla de bits se puede acceder sin necesidad de acceder a disco. Pero incluso con los tamaños de memoria de hoy, 4 Mbytes es una cantidad considerable de memoria principal para dedicar a una única función. La alternativa es poner la tabla de bits en disco. Pero una tabla de bits de 4 Mbytes requeriría alrededor de 8000 bloques de disco. No se puede hacer una búsqueda sobre esta cantidad de espacio de disco cada vez que se necesita un bloque, de tal forma que una tabla de bits residente en memoria es indicada.

Incluso cuando la tabla de bits está en memoria principal, una búsqueda exhaustiva de la tabla puede ralentizar el rendimiento del sistema de ficheros hasta un grado inaceptable. Esto es especialmente cierto cuando el disco está casi lleno y hay pocos bloques libres restantes. De forma análoga, la mayoría de los sistemas de ficheros que utilizan tablas de bits mantienen estructuras de datos auxiliares que resumen los contenidos de subrangos de la tabla de bits. Por ejemplo, la tabla se podía dividir lógicamente en varios subrangos de igual tamaño. Una tabla resumen podría incluir, para cada subrango, el número de bloques libres y el número de bloques libres contiguos de tamaño máximo. Cuando el sistema de ficheros necesita varios bloques contiguos, puede analizar la tabla resumen para encontrar un subrango apropiado y entonces buscar dentro de dicho subrango.

Porciones libres encadenadas

Las porciones libres se pueden encadenar utilizando un puntero y valor de longitud en cada porción libre. Este método tiene una sobrecarga de espacio insignificante, porque no se necesita una tabla de asignación de disco, sino simplemente un puntero al comienzo de la cadena y la longitud de la primera porción. Este método es apropiado para todos los métodos de asignación de ficheros. Si se asigna un

bloque cada vez, simplemente hay que escoger el bloque libre en la cabeza de la cadena y ajustar el primer puntero o el valor de la longitud. Si la asignación se hace de una porción de longitud variable, se debe utilizar un algoritmo de primer ajuste: se cargan las cabeceras de las porciones para determinar la siguiente porción libre apropiada en la cadena. De nuevo, se ajustan el puntero y los valores de longitud.

Este método tiene sus propios problemas. Después de cierto uso, el disco se quedará bastante fragmentado y muchas porciones serán de la longitud de un único bloque. También obsérvese que cada vez que se asigne un bloque, se necesita leer el bloque primero para recuperar el puntero al nuevo primer bloque libre antes de escribir datos en dicho bloque. Si se necesitan asignar muchos bloques individuales de una vez para una operación sobre ficheros, esto ralentiza en gran medida la creación de ficheros. Similarmente, borrar ficheros altamente fragmentados es una tarea que consume mucho tiempo.

Indexación

La técnica de indexación trata el espacio libre como un fichero y utiliza una tabla de índices tal y como se describió en la asignación de ficheros. Por motivos de eficiencia, el índice se debería utilizar en base a porciones de tamaño variable en lugar de bloques. Por tanto, hay una entrada en la tabla por cada porción libre en el disco. Esta técnica proporciona soporte eficiente a todos los métodos de asignación de ficheros.

Lista de bloques libres

En este método, a cada bloque se le asigna un número secuencialmente y la lista de los números de todos los bloques libres se mantiene en una porción reservada del disco. Dependiendo del tamaño del disco, se necesitarán 24 o 32 bits para almacenar un único número de bloque, de tal forma que el tamaño de la lista de bloques libres es 24 o 32 veces el tamaño de la correspondiente tabla de bits y por tanto debe almacenarse en disco y no en memoria principal. Sin embargo, este es un método satisfactorio. Considérense los siguientes puntos:

1. El espacio en disco dedicado a la lista de bloques libres es menor que el 1% del espacio total de disco. Si se utiliza un número de bloque de 32 bits, entonces la penalización de espacio es de 4 bytes por cada bloque de 512 bytes.
2. Aunque la lista de bloques libres es demasiado grande para almacenarla en memoria principal, hay dos técnicas efectivas para almacenar una pequeña parte de la lista en memoria principal.

   a) La lista se puede tratar como una pila (Apéndice 1B) con los primeros miles de elementos de la pila residentes en memoria principal. Cuando se asigna un nuevo bloque, se saca de la pila, que está en memoria principal. Similarmente, cuando se desasigna un bloque, se coloca en la pila. Cuando la porción de la pila en memoria se llena o se vacía, hay sólo una transferencia entre disco y memoria principal. Por tanto, esta técnica da acceso muy rápido en la mayoría de las ocasiones.

   b) La lista se puede tratar como una cola FIFO, con unos pocos miles de entradas desde la cabeza al final de la cola en memoria principal. Se asigna un bloque tomando la primera entrada de la cabeza de la cola y se desasigna añadiéndolo al final de la cola. Sólo hay una transferencia entre disco y memoria principal cuando la porción en memoria de la cabeza de la cola se vacía o la porción en memoria del final de la cola se llena.

En cualquiera de las estrategias listadas en el punto precedente (pila o cola FIFO), un hilo en segundo plano puede ordenar lentamente la lista en memoria o listas para facilitar la asignación contigua.

FIABILIDAD

Considérese el siguiente escenario:

1. El usuario A solicita una asignación de ficheros para añadir a un fichero existente.
2. La petición se concede y el disco y las tablas de asignación de ficheros se actualizan en memoria principal pero todavía no en disco.
3. El sistema falla y consecuentemente se reinicia.
4. El usuario B solicita una asignación de ficheros y se asigna espacio en disco que solapa la última asignación al usuario A.
5. El usuario A accede a una porción solapada a través de una referencia que se almacena dentro del fichero de A.

Esta dificultad surge debido al hecho de que el sistema mantiene una copia de la tabla de asignación de disco y la tabla de asignación de ficheros en memoria principal por motivos de eficiencia. Para prevenir este tipo de error, cuando se solicita una asignación de ficheros se pueden llevar a cabo los siguientes pasos:

1. Bloquear la tabla de asignación de disco en disco. Esto previene a otro usuario de causar alteraciones a la tabla hasta que esta asignación se complete.
2. Buscar espacio disponible en la tabla de asignación de disco. Esto supone que una copia de la tabla de asignación de disco siempre se guarda en memoria principal. Si no, debe primero traerse de disco.
3. Asignar espacio, actualizar la tabla de asignación de disco y actualizar el disco. Actualizar el disco supone escribir la tabla de asignación de disco en disco. Para la asignación de disco encadenado, también supone actualizar algunos punteros en disco.
4. Actualizar la tabla de asignación de disco y actualizar el disco.
5. Desbloquear la tabla de asignación de disco.

Esta técnica evitará errores. Sin embargo, cuando se asignan frecuentemente pequeñas porciones, el impacto en el rendimiento será substancial. Para reducir esta sobrecarga, se podría utilizar un esquema de asignación en lotes de almacenamiento. En este caso, se obtiene un lote de porciones libres en el disco para asignación. Las correspondientes porciones de disco se marcan «en uso». La asignación utilizando este lote puede realizarse en memoria principal. Cuando se finalice el lote, la tabla de asignación de disco se actualiza en disco y se puede adquirir un nuevo lote. Si ocurriera un fallo en el sistema, porciones del disco marcados «en uso» deben limpiarse de alguna forma antes de que se puedan reasignar. La técnica de limpieza dependerá de las características particulares del sistema de ficheros.

12.7. GESTIÓN DE FICHEROS DE UNIX

En el sistema de ficheros UNIX, se pueden distinguir seis tipos de ficheros:

- **Regulares u ordinarios.** Contiene datos arbitrarios en cero o más bloques de datos. Los ficheros regulares contienen información introducida por un usuario, una aplicación o una utilidad del sistema. El sistema de ficheros no impone ninguna estructura interna a un fichero regular sino que lo trata como una ristra de bytes.

- **Directorios.** Contiene una lista de nombres de ficheros más punteros a nodos-i asociados (nodos índice), descritos posteriormente. Los directorios se organizan jerárquicamente (Figura 12.4). Los directorios son realmente ficheros ordinarios con privilegios de protección de escritura especiales de tal forma que sólo el sistema de ficheros puede escribirlos, mientras que los programas de usuario tienen acceso de lectura.
- **Especiales.** No contienen datos, sino que proporcionan un mecanismo para asociar dispositivos físicos a nombres de ficheros. Se utilizan nombres de ficheros para acceder a los dispositivos periféricos, tales como terminales e impresoras. Cada dispositivo de E/S se asocia con un fichero especial, como se discutió en la Sección 11.8.
- **Tuberías con nombre.** Como se discutió en la Sección 6.7, una tubería es una utilidad de comunicación entre procesos. Una tubería guarda en un *buffer* los datos de su entrada de forma que un proceso que lea de la salida de la tubería reciba los datos del mismo modo que si leyera de una cola FIFO.
- **Enlaces.** En esencia, un enlace es un nombre alternativo de fichero para un fichero existente.
- **Enlaces simbólicos.** Se trata de un fichero de datos que contiene el nombre del fichero al que enlaza.

Esta sección trata la gestión de ficheros ordinarios, que corresponden a lo que la mayoría de los sistemas trata como ficheros.

NODOS-I

Todos los tipos de ficheros UNIX se administran por el sistema operativo mediante los nodos-i. Un nodo-i (nodo índice) es una estructura de control que contiene la información clave necesaria de un fichero particular para el sistema operativo. Varios nombres de ficheros se pueden asociar con un único nodo-i, pero un nodo-i activo se asocia con exactamente un fichero, y cada fichero es controlado por exactamente un nodo-i.

Los atributos del fichero así como sus permisos y otra información de control se almacenan en el nodo-i. La Tabla 12.4 lista los atributos de ficheros almacenados en el nodo-i de una implementación UNIX típica.

En el disco, hay una tabla de nodos-i, o lista de nodos-i, que contiene los nodos-i de todos los ficheros del sistema de ficheros. Cuando se abre un fichero, se trae su nodo-i a memoria principal y se almacena en una tabla de nodos-i residente en memoria.

ASIGNACIÓN DE FICHEROS

La asignación de ficheros se realiza a nivel de bloque. La asignación es dinámica, es decir, cuando se necesita, en lugar de utilizar preasignación. Por tanto, los bloques de un fichero en disco no son necesariamente contiguos. Se utiliza un método indexado para guardar traza de cada fichero, con parte del índice almacenado en el nodo-i del fichero. El nodo-i incluye 39 bytes de información de dirección que se organizan como trece direcciones de 3 bytes o punteros. Las primeras 10 direcciones apuntan a los primeros 10 bloques de datos del fichero. Si el fichero es mayor de 10 bloques, se utilizan uno o más niveles de indirección como se indica a continuación:

La dirección undécima en el nodo-i apunta a un bloque en disco que contiene la siguiente porción del índice. Esto se conoce como bloque indirecto simple. Este bloque contiene los punteros a siguientes bloques en el fichero.

Tabla 12.4. Información de un nodo-i UNIX residente en disco.

Modo de fichero	Espacio de 16 bits que almacena los permisos de acceso y ejecución asociados con el fichero.
	12-14 Tipo de fichero (regular, directorio, especial de caracteres o bloques, tubería FIFO)
	9-11 *Flags* de ejecución
	8 Permiso de lectura para el propietario
	7 Permiso de escritura para el propietario
	6 Permiso de ejecución para el propietario
	5 Permiso de lectura para el grupo del propietario
	4 Permiso de escritura para el grupo del propietario
	3 Permiso de ejecución para el grupo del propietario
	2 Permiso de lectura para el resto de usuarios
	1 Permiso de escritura para el resto de usuarios
	0 Permiso de ejecución para el resto de usuarios
Número de enlaces	Número de directorios que referencian a este nodo-i
Identificación del propietario	Propietario individual del fichero
Identificación de grupo del propietario	Grupo del propietario asociado con este fichero
Tamaño de fichero	Número de bytes del fichero
Direcciones de fichero	39 bytes de información de direcciones
Último acceso	Fecha del último acceso al fichero
Última modificación	Fecha de la última modificación del fichero
Nodo-i modificado	Fecha de la última modificación del nodo-i

Si el fichero contiene más bloques, la duodécima dirección del nodo-i apunta a un bloque indirecto doble. Este bloque contiene una lista de direcciones de bloques indirectos simples adicionales. Cada uno de los bloques indirectos simples, a su vez, contiene punteros a bloques de ficheros.

Si el fichero contiene todavía más bloques, la dirección decimotercera en el nodo-i apunta a un bloque indirecto triple que constituye un tercer nivel de indexación. Este bloque apunta a bloques indirectos dobles adicionales.

Todo esto se muestra en la Figura 12.13. La primera entrada del nodo-i contiene información sobre este fichero o directorio (Tabla 12.4). Las entradas restantes son las direcciones recién descritas. El número total de bloques de datos en el fichero depende de la capacidad de los bloques de tamaño fijo en el sistema. En UNIX System V, la longitud de un bloque es 1 Kbyte y cada bloque puede contener un total de 256 direcciones de bloque. Por tanto, el tamaño máximo de un fichero con este esquema es cerca de 16 Gbytes (Tabla 12.5).

Este esquema tiene varias ventajas:

1. El nodo-i es de tamaño fijo y relativamente pequeño y por tanto, se puede almacenar en memoria principal durante periodos largos.

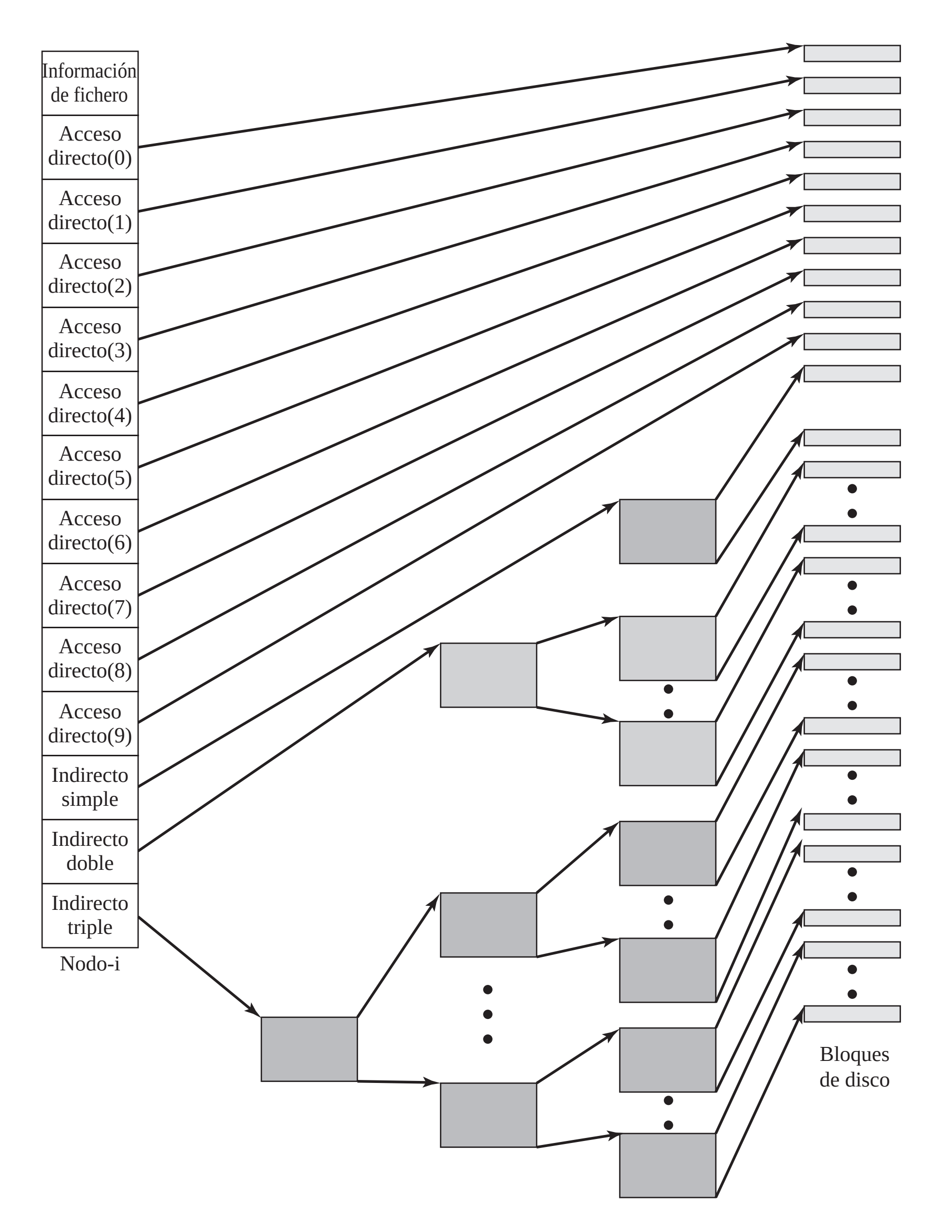


Figura 12.13. Disposición de un fichero UNIX en un disco.

Tabla 12.5. Capacidad de un fichero UNIX.

Nivel	Número de bloques	Número de bytes
Directo	10	10K
Simple indirecto	256	256K
Doble indirecto	256 x 256 = 65K	65M
Triple indirecto	256 x 65K = 16M	16G

2. A los ficheros más pequeños se puede acceder con poca o ninguna indirección, reduciendo el procesamiento y el tiempo de acceso a disco.
3. El tamaño máximo teórico de un fichero es suficientemente grande para satisfacer prácticamente todas las aplicaciones.

DIRECTORIOS

Los directorios se estructuran como un árbol jerárquico. Cada directorio contiene ficheros y/u otros directorios. Un directorio que se encuentra dentro de otro directorio se denomina subdirectorio. Como se mencionó anteriormente, un directorio es simplemente un fichero que contiene una lista de nombres de ficheros más punteros a nodos-i asociados. La Figura 12.14 muestra la estructura global. Cada entrada de directorio (entradaD) contiene un nombre para el fichero asociado o subdirectorio más un entero llamado el número-i (número índice). Cuando se accede al fichero o directorio, su número-i se utiliza como un índice en la tabla de nodos-i.

ESTRUCTURA DEL VOLUMEN

Un sistema de ficheros UNIX reside en un único disco lógico o partición de disco y se compone de los siguientes elementos:

- **Bloque de arranque.** Contiene el código requerido para arrancar el sistema operativo.
- **Superbloque.** Contiene atributos e información sobre el sistema de ficheros, tal como el tamaño de la partición y el tamaño de la tabla de nodos-i.
- **Tabla de nodos-i.** La colección de nodos-i para cada fichero.
- **Bloques de datos.** El espacio de almacenamiento disponible para los ficheros de datos y subdirectorios.

12.8. SISTEMA DE FICHEROS VIRTUAL LINUX

Linux incluye una utilidad versátil y potente para gestión de ficheros, diseñado para soportar una gran variedad de sistemas de gestión de ficheros y estructuras de ficheros. El enfoque usado en Linux consiste en hacer uso del **sistema de ficheros virtual (VFS)**, que presenta una única y uniforme interfaz de sistema de ficheros para los procesos de usuario. VFS define un modelo de ficheros común que es capaz de representar cualquier característica general y comportamiento de un sistema de ficheros concebible. VFS asume que los ficheros son objetos de un sistema de almacenamiento masivo del computador que comparten propiedades básicas sin tener en cuenta el sistema de ficheros concreto o el hardware subyacente. Los ficheros tienen nombres simbólicos que les permiten identificarse de forma única dentro de un directorio específico en el sistema de ficheros. Un fichero tiene un propietario, protección frente a accesos o modificaciones no autorizadas y otras propiedades. Un fichero se puede crear, leer, escribir o borrar. Para cualquier sistema de ficheros específico, se necesita un módulo de proyección que transforme las características del sistema de ficheros real a las características esperadas por el sistema de ficheros virtual.

La Figura 12.15 indica los ingredientes clave de la estrategia del sistema de ficheros Linux. Un proceso de usuario invoca una llamada al sistema de ficheros (por ejemplo, lectura) utilizando el esquema de ficheros VFS. VFS convierte esta llamada en una llamada al sistema de ficheros interno (del núcleo) que se pasa a una función de proyección del sistema de ficheros específico [por ejemplo,

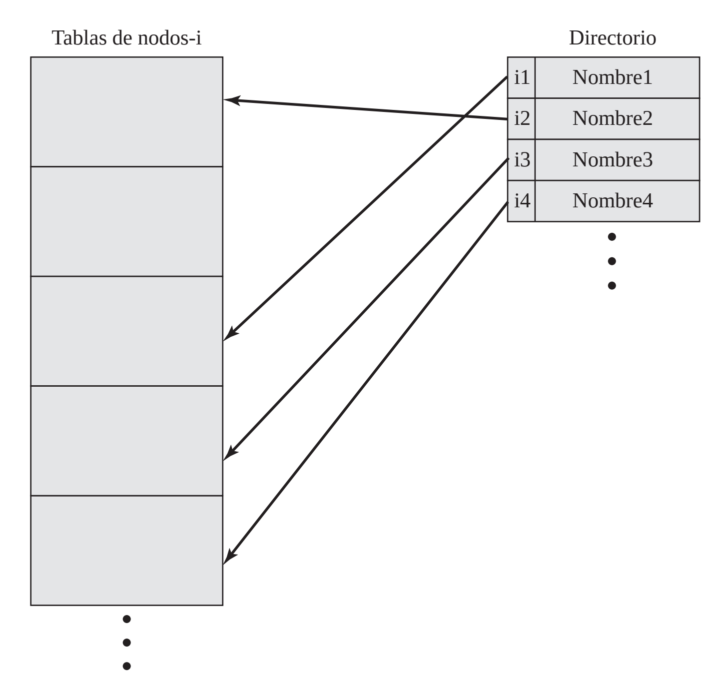


Figura 12.14. Directorios y nodos-i UNIX.

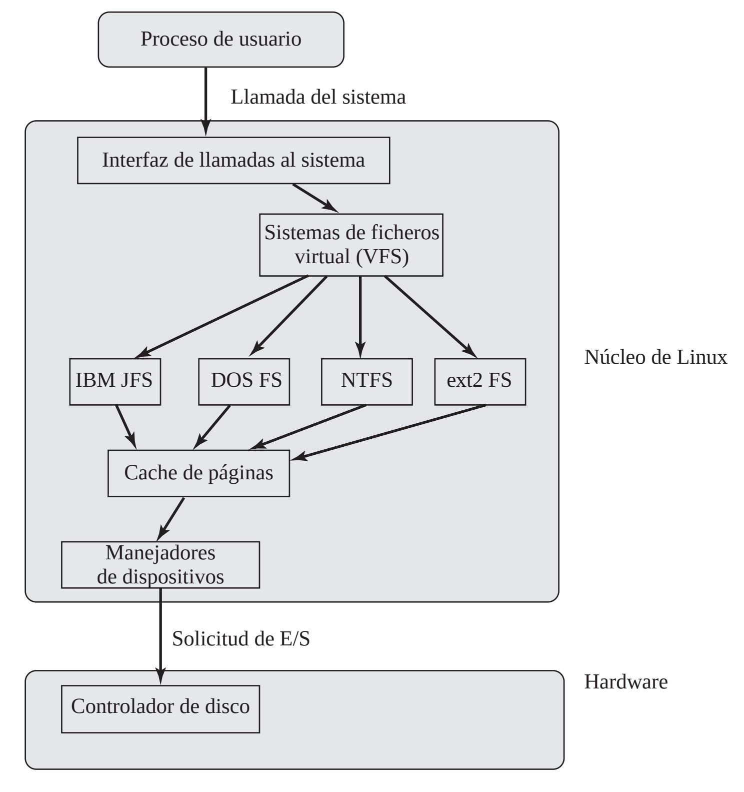


Figura 12.15. Contexto del sistema de ficheros virtual de Linux.

el sistema de ficheros JFS (*Journaling File System*) de IBM]. En la mayoría de los casos, la función de proyección es simplemente una proyección de las llamadas funcionales del sistema de ficheros desde un esquema a otro. En algunos casos, la función de proyección es más compleja. Por ejemplo, algunos sistemas de ficheros utilizan una tabla de asignación de ficheros (*File Allocation Table*, FAT), que almacena la posición de cada fichero en el árbol de directorios. En estos sistemas de ficheros, los directorios no son ficheros. En cualquier caso, la llamada original al sistema de ficheros se traduce en una llamada que es nativa para el sistema de ficheros destino. El software del sistema de ficheros destino es invocado entonces para llevar a cabo la función requerida sobre el fichero o directorio, bajo su control y almacenamiento secundario. Los resultados de la operación se comunican de nuevo al usuario de una forma similar.

La Figura 12.16 indica el papel que VFS juega dentro del núcleo de Linux. Cuando un proceso inicia una llamada al sistema orientada a ficheros (por ejemplo, lectura), el núcleo llama a una función en VFS. Esta función gestiona los aspectos independientes del sistema de ficheros e inicia una llamada a una función en el código del sistema de ficheros destino. Esta llamada pasa a través de una función de proyección que convierte la llamada VFS en una llamada al sistema de ficheros destino. VFS es independiente de cualquier sistema de ficheros, de modo que la implementación de una función de proyección debe ser parte de la implementación de un sistema de ficheros en Linux. El sistema de ficheros destino convierte la petición del sistema de ficheros en instrucciones orientadas a dispositivo que se pasan al controlador del dispositivo mediante funciones de *cache* de páginas.

VFS es un esquema orientado a objetos. Debido a que está escrito en C, en lugar de en un lenguaje que dé soporte a la programación de objetos (como C++ o Java), los objetos VFS se implementan simplemente como estructuras de datos C. Cada objeto contiene tanto datos como punteros a las funciones implementadas del sistema de ficheros que operan sobre los datos. Los cuatro tipos de objetos primarios en VFS son los siguientes:

- **Objeto superbloque.** Representa un sistema de ficheros montado específico.
- **Objeto nodo-i.** Representa un fichero específico.
- **Objeto entrada de directorio.** Representa una entrada de directorio específica.
- **Objeto de fichero.** Representa un fichero abierto asociado con un proceso.

Este esquema se basa en los conceptos utilizados en el sistema de ficheros de UNIX, tal y como se describió en la Sección 12.7. Los conceptos clave del sistema de ficheros de UNIX a recordar son los siguientes. Un sistema de ficheros está compuesto por una organización jerárquica de directorios.

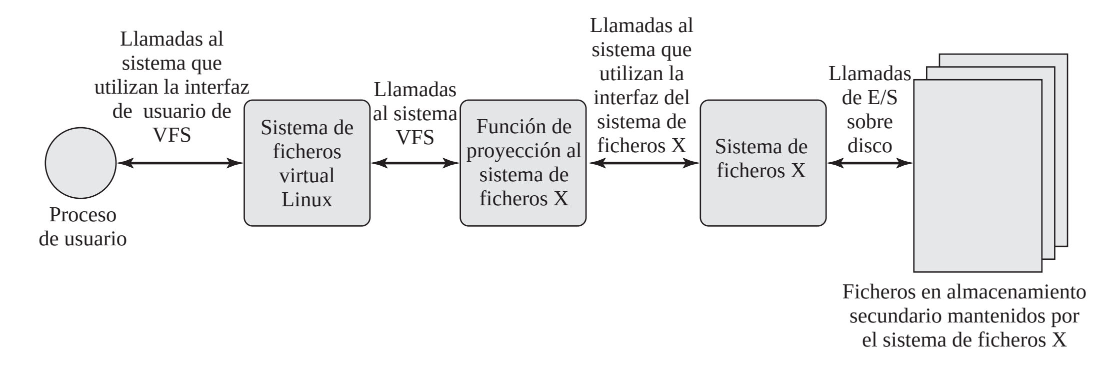


Figura 12.16. Concepto del sistema de ficheros virtual de Linux.

Un directorio es análogo a una carpeta y puede contener ficheros y/u otros directorios. Debido a que un directorio puede contener otros directorios, se forma una estructura de árbol. Un camino a través de la estructura de árbol desde la raíz está formado por una secuencia de entradas de directorio, acabando en una entrada de directorio (entradaD) o un nombre de fichero. En UNIX, un directorio se implementa como un fichero que lista los ficheros y directorios contenidos en él. Por tanto, las operaciones de ficheros se pueden llevar a cabo tanto sobre ficheros como directorios.

EL OBJETO SUPERBLOQUE

El objeto superbloque almacena información que describe un sistema de ficheros específico. Típicamente, el superbloque corresponde al superbloque del sistema de ficheros o bloque de control del sistema de ficheros, que se almacena en un sector especial en el disco.

El objeto superbloque está formado por varios elementos de datos. Ejemplos de estos elementos incluyen los siguientes:

- El dispositivo sobre el cual el sistema de ficheros está montado.
- El tamaño de bloque básico del sistema de ficheros.
- El *dirty flag*, que indica que se ha cambiado el superbloque pero no se ha escrito a disco.
- Tipo de fichero.
- *Flags*, como el de sólo lectura.
- Puntero al directorio raíz del sistema de ficheros.
- Lista de ficheros abiertos.
- Semáforo para controlar el acceso al sistema de ficheros.
- Lista de operaciones de superbloque.

El último elemento en la lista precedente se refiere a un objeto de operaciones contenido en el objeto superbloque. El objeto operación define los métodos de objeto (funciones) que el núcleo puede invocar sobre el objeto superbloque. Los métodos definidos para el objeto superbloque incluyen las siguientes:

- *** read_inode.** Leer un nodo-i específico desde un sistema de ficheros montado.
- * **write_inode.** Escribir un nodo-i dado a disco.
- * **put_inode.** Obtener un nodo-i.
- * **delete_inode.** Borrar un nodo-i del disco.
- * **notify_change.** Invocado cuando se cambian los atributos.
- * **put_super.** Llamado por VFS cuando el sistema de ficheros está desmontado, para obtener el superbloque dado.
- * **write_super.** Invocado cuando VFS decide que el superbloque necesita escribirse en disco.
- * **statfs.** Obtiene las estadísticas del sistema de ficheros.
- * **remount_fs.** Llamado por VFS cuando el sistema de ficheros es montado de nuevo con nuevas opciones de montaje.
- * **clear_inode.** Obtener nodo-i y borrar cualquier página que contenga datos relacionados.

EL OBJETO NODO-I

Un nodo-i está asociado con cada fichero. El objeto nodo-i contiene toda la información sobre un determinado fichero excepto su nombre y el contenido real del fichero. Los elementos contenidos en un objeto nodo-i incluyen el propietario, grupo, permisos, tiempos de acceso para un fichero, tamaño de los datos que contiene y número de enlaces.

El objeto nodo-i también incluye un objeto operaciones de nodo-i, que describe las funciones implementadas en el sistema de ficheros que VFS puede invocar sobre el nodo-i. Los métodos definidos por el objeto nodo-i incluyen los siguientes:

- **create.** Crear un nodo-i nuevo para un fichero regular asociado con un objeto entradaD en algún directorio.
- **lookup.** Buscar en un directorio el nodo-i correspondiente a un nombre de fichero.
- **mkdir.** Crear un nuevo nodo-i para un directorio asociado a un objeto entradaD en algún directorio.

EL OBJETO ENTRADAD

Una entradaD (entrada de directorio) es un componente específico de una ruta. El componente puede ser un nombre de directorio o un nombre de fichero. El objeto entradaD facilita el acceso a los ficheros y directorios y se utilizan en una *cache* de entradaD para dicho propósito.

EL OBJETO FICHERO

El objeto fichero se utiliza para representar un fichero abierto por un proceso. El objeto se crea en respuesta a la llamada al sistema *open()* y se destruye en respuesta a la llamada al sistema *close()*. El objeto fichero está formado por un conjunto de elementos, que incluye los siguientes:

- Objeto entradaD asociado con el fichero.
- Sistema de ficheros que contiene el fichero.
- Contador de uso del objeto fichero.
- Identificador del usuario.
- Identificador de grupo del usuario.
- Puntero de posición del fichero, que es la posición actual en el fichero desde la cual tendrá lugar la siguiente operación.

El objeto fichero también incluye un objeto operaciones de nodo-i, que describe las funciones implementadas del sistema de ficheros que VFS puede invocar sobre el objeto fichero. Los métodos definidos para el objeto fichero incluyen la lectura, escritura, apertura, creación y bloqueo.

12.9. SISTEMA DE FICHEROS DE WINDOWS

Windows da soporte a varios sistemas de ficheros, incluyendo el sistema FAT (*File Allocation Table*: tabla de asignación de ficheros) que ejecutan en Windows 95, MS-DOS y OS/2. Pero los desarrolla-

dores de Windows también diseñaron un nuevo sistema de ficheros, el sistema de ficheros de Windows (NTFS), que está pensado para alcanzar requisitos de altas prestaciones en estaciones de trabajo y servidores. Ejemplos de aplicaciones de altas prestaciones incluyen las siguientes:

- Aplicaciones cliente/servidor tales como los servidores de ficheros, servidores de computación y servidores de bases de datos.
- Ingeniería intensiva de recursos y aplicaciones científicas.
- Aplicaciones de red para grandes sistemas corporativos.

CARACTERÍSTICAS CLAVE DE NTFS

NTFS es un sistema de ficheros flexible y potente, construido, como se verá, en un modelo de sistema de ficheros elegantemente simple. Las características más notables de NTFS incluyen las siguientes:

- **Recuperación.** Uno de los requisitos más importantes del nuevo sistema de ficheros Windows es la capacidad de recuperarse frente a errores en el sistema y los fallos de disco. En el caso de dichos fallos, NTFS es capaz de reconstruir volúmenes de disco y devolverlos a un estado consistente. Esto se lleva a cabo utilizando un modelo de procesamiento de transacciones para los cambios en el sistema de ficheros; cada cambio significativo se trata como una acción atómica que se realiza de forma completa o no se lleva a cabo en absoluto. Cada transacción que está en proceso cuando se produce un fallo es a continuación terminada completamente o bien se deja el sistema como antes de su ejecución. Adicionalmente, NTFS utiliza almacenamiento redundante para datos del sistema de ficheros críticos, de forma que un fallo en el sector de un disco no cause la pérdida de datos que describen la estructura y estado del sistema de ficheros.
- **Seguridad.** NTFS utiliza el modelo de objetos de Windows para forzar la seguridad. Un fichero abierto se implementa como un objeto fichero con un descriptor de seguridad que define sus atributos de seguridad.
- **Discos y ficheros grandes.** NTFS soporta discos y ficheros muy grandes de forma más eficiente que la mayoría del resto de los sistemas de ficheros, incluyendo FAT.
- **Múltiples flujos de datos.** Los contenidos reales de un fichero se tratan como un flujo de bytes. En NTFS es posible definir múltiples flujos de datos para un único fichero. Un ejemplo de la utilidad de esta característica es que permite que sistemas Macintosh remotos utilicen Windows para almacenar y recuperar ficheros. En Macintosh, cada fichero tiene dos componentes: los datos del fichero y un contenedor de recursos que tiene información sobre el fichero. NTFS trata estos dos componentes como dos flujos de datos.
- **Facilidad general de indexación.** NTFS asocia una colección de atributos con cada fichero. El conjunto de descripciones de fichero en el sistema de gestión de ficheros se organiza como una base de datos relacional, de forma que los ficheros se pueden indexar por cualquier atributo.

VOLÚMENES NTFS Y ESTRUCTURA DE FICHEROS

NTFS hace uso de los siguientes conceptos de almacenamiento de disco:

- **Sector.** La unidad física de almacenamiento más pequeña en el disco. El tamaño de los datos en bytes es una potencia de 2 y es casi siempre 512 bytes.

- **Agrupación.** Uno o más sectores contiguos (próximos entre sí en la misma pista). El tamaño de la agrupación en sectores es una potencia de 2.
- **Volumen.** Una partición lógica de un disco, formada por una o más agrupaciones y utilizada por un sistema de ficheros para asignar espacio. En cualquier momento, un volumen está formado por información del sistema de ficheros, una colección de ficheros y cualquier espacio restante adicional sin asignar del volumen que se puede asignar a los ficheros. Un volumen puede ser todo o parte de un único disco o se puede extender entre múltiples discos. Si se emplea RAID 5 hardware o software, un volumen está compuesto por tiras a través de múltiples discos. El tamaño máximo de un volumen en NTFS es de 2^{64} bytes.

La agrupación es la unidad fundamental de asignación en NTFS, el cual no reconoce sectores. Por ejemplo, supóngase que cada sector es 512 bytes y el sistema se configura con dos sectores por agrupación (una agrupación = 1K bytes). Si un usuario crea un fichero de 1600 bytes, se asignan dos agrupaciones al fichero. Posteriormente, si el usuario actualiza el fichero a 3200 bytes, se asignan otras dos agrupaciones. Las agrupaciones asignadas a un fichero no necesitan ser contiguas; es posible fragmentar un fichero en el disco. Actualmente, el tamaño de fichero máximo soportado por NTFS es 2^{32} agrupaciones, lo que equivale a un máximo de 2^{48} bytes. Una agrupación puede tener como máximo 2^{16} bytes.

El uso de agrupaciones para la asignación hace a NTFS independiente del tamaño de sector físico. Esto habilita a NTFS a soportar fácilmente discos no estándares que no tengan un tamaño de sector de 512 bytes y a soportar eficientemente discos y ficheros muy grandes utilizando un tamaño de agrupación más grande. La eficiencia procede del hecho de que los sistemas de ficheros deben guardar traza de cada agrupación asignada a cada fichero; con agrupaciones mayores, hay menos elementos que gestionar.

La Tabla 12.6 muestra los tamaños de agrupación por omisión para NTFS. Los tamaños por omisión dependen del tamaño del volumen. El tamaño de agrupación que se utiliza para un volumen particular lo establece NTFS cuando el usuario solicita que se formatee un volumen.

Tabla 12.6. Particiones y tamaños de agrupaciones de Windows NTFS.

Tamaño de volumen	Sectores por agrupación	Tamaño de agrupación
≤ 512 Mbytes	1	512 bytes
512 Mbytes - 1 Gbyte	2	1K
1 Gbyte – 2 Gbytes	4	2K
2 Gbytes – 4 Gbytes	8	4K
4 Gbytes – 8 Gbytes	16	8K
8 Gbytes – 16 Gbytes	32	16K
16 Gbytes – 32 Gbytes	64	32K
> 32 Gbytes	128	64K

ESTRUCTURA DE UN VOLUMEN NTFS

NTFS utiliza un enfoque notablemente simple pero potente para organizar información de un volumen en el disco. Cada elemento del volumen es un fichero, y cada fichero está formado por una colección de atributos. Incluso el contenido de un fichero se trata como un atributo. Con esta estructura sencilla, unas pocas funciones de propósito general son suficientes para organizar y gestionar un sistema de ficheros.

La Figura 12.17 muestra la estructura de un volumen NTFS, que está formado por cuatro regiones. Los primeros sectores de un volumen están ocupados por el **sector de arranque de la partición** (aunque se llame sector, puede estar formado por un máximo de 16 sectores), que contiene información sobre la estructura del volumen, las estructuras del sistema de ficheros así como la información de arranque de inicio y el código. Esto es seguido por la **tabla maestra de ficheros** (*Master File Table*, MFT), que contiene información sobre todos los ficheros y carpetas (directorios) de este volumen NTFS así como la información sobre espacio disponible no asignado. En esencia, el MFT es una lista de todos los contenidos de este volumen NTFS, organizada como un conjunto de filas en una estructura de base de datos relacional.

Siguiendo al MFT hay una región, típicamente de 1 Mbyte de longitud, conteniendo los **ficheros del sistema**. Entre los ficheros de esta región se encuentran los siguientes:

- **MFT2.** Un espejo de las tres primeras filas del MFT, utilizado para garantizar el acceso al MFT en el caso de un fallo de un único sector.
- **Fichero registro.** Una lista de pasos de transacciones utilizadas para la recuperación en NTFS.
- **Mapa de bits de las agrupaciones.** Una representación del volumen, mostrando qué agrupaciones están en uso.
- **Tabla de definición de atributos.** Define los tipos de atributos soportados en este volumen e indica si se pueden indexar o si se pueden recuperar durante una operación de recuperación del sistema.

Tabla maestra de ficheros

El corazón del sistema de ficheros Windows es el MFT. El MFT se organiza como una tabla de filas de longitud variable, llamadas registros. Cada fila describe un fichero o una carpeta de este volumen, incluyendo el propio MFT, que se trata como un fichero. Si un fichero es suficientemente pequeño, el fichero completo se localiza en una fila del MFT. En otro caso, la fila para dicho fichero contiene información parcial y el resto del fichero se encuentra en otras agrupaciones disponibles del volumen. Un puntero a dichas agrupaciones se encuentra en la fila MFT del fichero.

Cada registro del MFT está formado por un conjunto de atributos que sirve para definir las características del fichero (o carpeta) y los contenidos del mismo. La Tabla 12.7 lista los atributos que se pueden encontrar en una fila, habiéndose sombreado los atributos obligatorios.

RECUPERACIÓN

NTFS hace posible llevar el sistema de ficheros a un estado consistente cuando ha habido un error de sistema o un fallo de disco. Los elementos clave que soportan la recuperación son los siguientes (Figura 12.18):

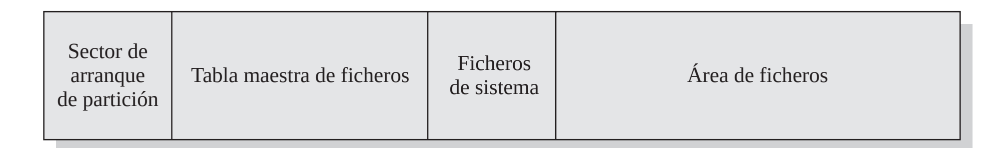


Figura 12.17. Disposición del volumen de NTFS.

Tabla 12.7. Tipos de atributos de ficheros y directorios de Windows NTFS.

Tipo de atributos	Descripción
Información estándar	Incluye los atributos de acceso (sólo lectura, lectura y escritura, etc.); sellos de tiempo, incluyendo la fecha de creación y última modificación; y cuantos directorios apuntan al fichero (número de enlaces).
Lista de atributos	Una lista de atributos que componen el fichero y la referencia de ficheros del registro de ficheros MFT en el que se localiza cada atributo. Utilizado cuando todos los atributos no caben en un único registro de ficheros MFT.
Nombre de fichero	Un fichero o directorio debe tener uno o más nombres
Descriptor de seguridad	Especifica quién es el propietario del fichero y quién puede acceder a él.
Datos	Los contenidos del fichero. Un fichero tiene un conjunto de atributos sin nombre por omisión y podría tener uno o más atributos de datos con nombre.
Raíz índice	Usado para implementar carpetas
Asignación índice	Usado para implementar carpetas
Información de volumen	Incluye información relacionada con el volumen, tal como la versión y el nombre del volumen.
Mapa de bits	Proporciona un mapa que representa registros en uso del MFT o carpeta

Nota: Las filas coloreadas se refieren a los atributos de ficheros requeridos; los otros atributos son opcionales.

- **Gestor de E/S.** Incluye el controlador NTFS, que gestiona las funciones básicas de apertura, cierre, lectura y escritura de NTFS. Adicionalmente, el módulo software de RAID FT-DISK se puede configurar para su uso.
- **Servicio de fichero de registro.** Mantiene un registro de las escrituras de disco. El fichero de registro se utiliza para recuperar un volumen formateado NTFS en el caso de un fallo del sistema.
- **Gestor de cache.** Responsable del *caching* de las lecturas y escrituras de fichero para incrementar el rendimiento. El gestor de *cache* optimiza la E/S de disco utilizando escritura diferida y técnicas de transacciones diferidas, descritas en la Sección 11.8.
- **Gestor de memoria virtual.** NTFS accede a ficheros en *cache* mediante la proyección de las referencias al fichero a las referencias a memoria virtual, leyendo y escribiendo en la memoria virtual.

Es importante observar que los procedimientos de recuperación utilizados por NTFS se diseñan para recuperar los datos del sistema de ficheros, no los contenidos del fichero. Por tanto, debido a un error, el usuario nunca debería perder un volumen o la estructura del directorio/fichero. Sin embargo, el sistema de ficheros no garantiza los datos del usuario. Proporcionar recuperación completa, incluyendo los datos de usuario, implicaría el uso de una facilidad de recuperación más elaborada y que supone un mayor tiempo de ejecución.

La esencia de la capacidad de recuperación de NTFS se encuentra en el uso de registros (*logging*). Cada operación que altera un sistema de ficheros se trata como una transacción. Cada subope-

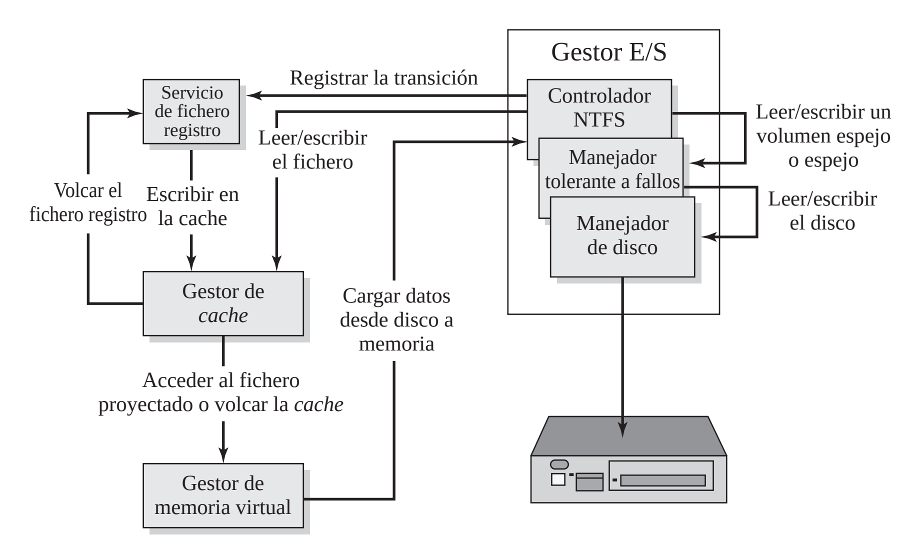


Figura 12.18. Componentes de Windows NTFS.

ración de una transacción que altera estructuras de datos del sistema de ficheros importantes se graba en un fichero de registro antes de grabarse en el volumen del disco. Utilizando el registro, una transacción parcialmente completada en el momento del error se puede rehacer posteriormente o deshacer cuando el sistema se recupera.

En términos generales, estos son los pasos tomados para asegurar la recuperación, como se describe en [CUST94]:

1. NTFS primero llama al registro del sistema de ficheros para grabar en el registro de la cache cualquier transacción que modificará la estructura del volumen.
2. NTFS modifica el volumen (en la cache).
3. El gestor de la cache llama al registro del sistema de ficheros para volcar el fichero registro al disco.
4. Una vez que el registro se actualiza de forma segura en el disco, el gestor de cache vuelca los cambios al disco.

12.10. RESUMEN

Un sistema de gestión de ficheros es un conjunto de software de sistema que proporciona servicios a usuarios y aplicaciones en el uso de ficheros, incluyendo accesos a ficheros, mantenimiento de directorios y control de acceso. El sistema de gestión de ficheros se ve típicamente como un servicio del sistema que el sistema operativo sirve, en lugar de ser parte del sistema operativo en sí. Sin embargo, en cualquier sistema, al menos parte de las funciones de gestión de ficheros se lleva a cabo por el sistema operativo.

Un fichero está formado por una colección de registros. La forma en la que estos registros se acceden determina su organización lógica, y hasta cierto punto su organización física en disco. Si un fichero se procesa primariamente como una entidad, la organización secuencial es la más sencilla y la más apropiada. Si se necesita acceso secuencial, pero también se desea acceso aleatorio al fichero in-

dividual, un fichero secuencial indexado puede proporcionar el mejor rendimiento. Si los accesos al fichero son principalmente aleatorios, un fichero indexado o *hash* puede ser el más apropiado.

Sea cual sea la estructura del fichero escogida, se necesita un servicio de directorios. Esto permite que los ficheros se organicen de forma jerárquica. Esta organización es útil para que el usuario guarde traza de los ficheros y es útil para que los sistemas de gestión de ficheros proporcionen control de acceso y otros servicios al usuario.

Los registros de los ficheros, incluso si son de tamaño fijo, generalmente no están conformes al tamaño de un bloque de disco físico. Por tanto, se necesita alguna especie de estrategia de asignación de bloques. Un compromiso entre la complejidad, el rendimiento y la utilización del espacio determina la estrategia de asignación de bloques a utilizar.

Una función clave de cualquier esquema de gestión de ficheros es la gestión del espacio de disco. Parte de esta función es la estrategia para asignar bloques de disco a un fichero. Varios métodos pueden utilizarse, y una gran variedad de estructuras de datos se utilizan para guardar traza de la asignación para cada fichero. Adicionalmente, se debe gestionar el espacio en disco que no se ha asignado. Esta última función consiste principalmente en mantener una tabla de asignación de disco que indique qué bloques están libres.

12.11. LECTURAS RECOMENDADAS

Existen varios libros buenos sobre gestión de ficheros. Todos ellos se enfocan en sistemas de gestión de ficheros, pero también tratan aspectos relacionados con el sistema operativo. Quizás el más útil es [WIED87], que sigue un enfoque cuantitativo para la gestión de ficheros y trata todos los aspectos descritos en la Figura 12.2, desde la planificación de disco a la estructura de ficheros. [LIVA90] hace hincapié en las estructuras de los ficheros, proporcionando una buena y abundante revisión de análisis de rendimiento comparativos. [GROS86] proporciona una visión equilibrada en aspectos relacionados con los métodos de E/S de ficheros y métodos de acceso a ficheros. También contiene una descripción general de todas las estructuras de control necesitadas por un sistema de ficheros. Éstas proporcionan una lista útil para el diseño de los sistemas de ficheros. [FOLK98] enfatiza el procesamiento de los ficheros, tratando los temas de mantenimiento, búsqueda, ordenación y compartición.

El sistema de ficheros Linux se examina detalladamente en [LOVE04] y [BOVE03]. Una buena descripción se recoge en [RUBI97].

[CUST94] proporciona una buena descripción del sistema de ficheros NT. [NAGA97] cubre el material más detalladamente.

BOVE03 Bovet, D., and Cesati, M. *Understanding the Linux Kernel.* Sebastopol, CA: O'Reilly, 2003.

CUST94 Custer, H. *Inside the Windows NT File System.* Redmond, WA: Microsoft Press, 1994.

FOLK98 Folk, M., and Zoellick, B. *File Structures: An Object-Oriented Approach with* C++. Reading, MA:Addison-Wesley, 1998.

GROS86 Grosshans,D. *File Systems: Design and Implementation.* Englewood Cliffs, NJ: Prentice Hall, 1986.

LIVA90 Livadas, P. *File Structures: Theory and Practice.* Englewood Cliffs, NJ: Prentice Hall, 1990.

LOVE04 Love, R. *Linux Kernel Development.* Indianapolis, IN: Sams Publishing, 2004.

NAGA97 Nagar, R. *Windows NT File System Internals.* Sebastopol, CA: O'Reilly, 1997.

RUBI97 Rubini, A. «The Virtual File System in Linux.» *Linux Journal*, May 1997.

WIED87 Wiederhold, G. *File Organization for Database Design.* New York: McGraw-Hill, 1987.

12.12. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

asignación de fichero	directorio de trabajo o actual	pila
asignación de fichero indexado	fichero	registro
asignación de ficheros contigua	fichero de acceso directo o *hash*	ruta del nombre
asignación de ficheros encadenada	fichero indexado	sistema de gestión de ficheros
base de datos	fichero secuencial	tabla de asignación de disco
bloque	fichero secuencial indexado	tabla de asignación de fichero
campo	método de acceso	tabla de bits
campo clave	nodo-i	
directorio	nombre de fichero	

CUESTIONES DE REPASO

12.1. ¿Cuál es la diferencia entre un campo y un registro?

12.2. ¿Cuál es la diferencia entre un fichero y una base de datos?

12.3. ¿Qué es un sistema de gestión de ficheros?

12.4. ¿Qué criterios son importantes a la hora de escoger una organización de ficheros?

12.5. Liste y defina brevemente cinco organizaciones de ficheros.

12.6. ¿Por qué es menor el tiempo de búsqueda medio de un registro en un fichero para un fichero secuencial indexado que para un fichero secuencial?

12.7. ¿Cuáles son las operaciones típicas que se pueden realizar sobre un directorio?

12.8. ¿Cuál es la relación entre una ruta de un fichero y un directorio de trabajo?

12.9. ¿Cuáles son los derechos de acceso típicos que se pueden conceder o denegar a un usuario particular sobre un fichero particular?

12.10. Liste y defina brevemente tres tipos de bloques utilizados.

12.11. Liste y defina brevemente tres métodos de asignación de ficheros.

PROBLEMAS

12.1. Defina:

B = tamaño de bloque

R = tamaño de registro

P = tamaño de puntero de bloque

F = factor de bloques; número esperado de registros dentro de un bloque

Defina una fórmula para F de acuerdo a los tres tipos de bloques dibujados en la Figura 12.6.

12.2. Un esquema para evitar el problema de la preasignación frente al malgasto o falta de contigüidad es asignar porciones de tamaño incremental a medida que el fichero crece. Por ejemplo, comenzando con un tamaño de un bloque y doblando la porción por cada asignación. Considérese un fichero con n registros con un factor de bloques F. Supóngase que se utiliza un índice simple de un nivel tal como una tabla de asignación de ficheros

a) Obtenga un límite superior del número de entradas en la tabla de asignación de ficheros como una función de F y n.

b) ¿Cuál es la cantidad máxima de espacio que no se asigna en ningún momento?

12.3. ¿Qué organizaciones de fichero escogería para maximizar la eficiencia en términos de velocidad de acceso, uso de espacio de almacenamiento y facilidad de actualización (añadir/borrar/modificar) cuando los datos:

a) no se actualizan frecuentemente y se acceden frecuentemente en orden aleatorio?

b) se actualizan frecuentemente y se acceden en su totalidad relativamente frecuente?

c) se actualizan frecuentemente y se acceden frecuentemente en orden aleatorio?

12.4. Los directorios se pueden implementar como «ficheros especiales» que sólo se pueden acceder de una forma limitada o como ficheros de datos ordinarios. ¿Cuáles son las ventajas y desventajas de cada opción?

12.5. Algunos sistemas operativos tienen un sistema de ficheros estructurado en forma de árbol, pero limitan la profundidad del árbol hasta un número pequeño de niveles. ¿Qué efecto tiene este límite sobre los usuarios? ¿Cómo simplifica esto el diseño del sistema operativo (si lo hace)?

12.6. Considérese un sistema de ficheros jerárquico en el que el espacio de disco libre se guarda en una lista de espacio libre.

a) Supóngase que se pierde el puntero al espacio libre. ¿Puede el sistema reconstruir la lista de espacio libre?

b) Sugiera un esquema que asegure que el puntero nunca se pierde como resultado de un único fallo de memoria.

12.7. Considérese la organización de un fichero UNIX representado por el nodo-i (Figura 12.13). Asúmase que hay 12 punteros directos a bloque, un puntero indirecto simple, otro doble y otro triple en cada nodo-i. Asúmase también que el tamaño de bloque del sistema y el tamaño de sector del disco son ambos de 8K. Si el puntero a bloque es de 32 bits, con 8 bits utilizados para identificar el disco físico y 24 bits para identificar el bloque físico, contestar a las siguientes preguntas:

a) ¿Cuál es el tamaño máximo de fichero soportado por este sistema?

b) ¿Cuál es el tamaño máximo de partición soportado por este sistema?

c) Asumiendo que sólo se conoce que el nodo-i del fichero está ya en memoria principal, ¿cuántos accesos a disco se requieren para acceder al byte de la posición 13.423.956?

PARTE VI

SISTEMAS DISTRIBUIDOS Y SEGURIDAD

Tradicionalmente, la función de procesamiento de datos estaba organizada de forma centralizada. En una arquitectura de procesamiento de datos centralizada, el tratamiento de datos se realizaba en un computador o un *cluster* de computadores, en general, grandes computadores, localizados en una instalación central de procesamiento de datos. Muchas de las tareas realizadas en esta instalación se inician con resultados producidos en la propia instalación. Un posible ejemplo es una aplicación de nóminas. Otras tareas podrían requerir acceso interactivo de personal que no está físicamente en el centro de procesamiento de datos. Por ejemplo, una función de entrada de datos, como una actualización del inventario, se puede realizar por personal en cualquier parte de la organización. En una arquitectura centralizada, cada persona tiene un terminal local que se conecta por un mecanismo de comunicaciones a la instalación central de procesamiento de datos.

Una instalación totalmente centralizada de procesamiento de datos, es centralizada en muchos sentidos:

- **Computadores centralizados.** En la instalación central hay uno o más computadores. En muchos casos, hay uno o dos grandes computadores, que requieren instalaciones especiales como aire acondicionado o suelos elevados. En una organización más pequeña, el computador o computadores centrales son grandes minicomputadores o sistemas de gama media. Los *iSeries* de IBM son un ejemplo de sistemas de gama media.
- **Procesamiento centralizado.** Todas las aplicaciones se ejecutan en las instalaciones centrales de procesamiento de datos. Esto incluye a aplicaciones claramente centrales y a aplicaciones no centrales por naturaleza, tales como una aplicación de nóminas o una aplicación que necesite usuarios en un determinado departamento organizativo. Un ejemplo de esta última situación es un departamento de diseño de productos que quiere utilizar un paquete gráfico de diseño asistido por computador (CAD) que ejecuta en las instalaciones centrales.
- **Datos centralizados.** Todos los datos se almacenan en ficheros y bases de datos en las instalaciones centrales y son controlados y accedidos por el computador o computadores centrales. Esto incluye tanto a datos que se utilizan en muchos departamentos de la organización, tales como figuras de inventario, como a datos que deberían ser sólo usados por un departamento. Un ejemplo de esta última situación, el departamento de *marketing* podría tener una base de datos con información derivada de encuestas de clientes.

Una organización centralizada de este tipo tiene varios aspectos atractivos. Podría haber una reducción de los costos en la compra y manejo de equipamiento y de software. Un gran departamento central de procesamiento de datos se puede permitir tener programadores profesionales en plantilla para asegurar las necesidades de varios departamentos. La dirección puede controlar la realización del procesamiento de datos, forzar estándares de programación y estructuras de ficheros de datos y diseñar e implementar una política de seguridad.

Una instalación central de procesamiento de datos se podría dividir de diversas maneras, implementando una estrategia de procesamiento de datos distribuida (*distributed data processing*, DDP). Una instalación distribuida de procesamiento de datos es aquélla en que los computadores, normalmente pequeños computadores, están dispersos por toda la organización. El objetivo de esta dispersión es procesar la información de la forma más eficiente posible, basándose en consideraciones operacionales, económicas y/o geográficas. Una instalación DDP podría incluir una instalación central además de instalaciones satélite, o podría estar formada por una serie de instalaciones similares. En cualquier caso, normalmente se necesita algún tipo de interconexión; es decir, los computadores del sistema deben estar conectados entre sí. Como se podría esperar, una instalación DDP involucra la distribución de computadores, procesamiento y datos.

Entre las ventajas de DDP se incluyen las siguientes:

- **Receptividad.** Las instalaciones de computación local se pueden gestionar de tal manera que pueden satisfacer más directamente las necesidades de gestión de las organizaciones locales que una instalación central, que intenta satisfacer las necesidades de toda la organización.
- **Disponibilidad.** Con múltiples sistemas interconectados, la pérdida de un sistema podría tener un impacto mínimo. Los sistemas y componentes prioritarios (por ejemplo, computadores con aplicaciones críticas, impresoras y dispositivos de almacenamiento masivo) podrían estar replicados de forma que un sistema de recuperación puede asumir la carga rápidamente después de un fallo.
- **Compartición de resursos.** El hardware más caro se puede compartir entre los usuarios. Los ficheros de datos se pueden gestionar y mantener de forma centralizada, pero con acceso a toda la organización. Los servicios para el personal, los programas y las bases de datos se pueden desarrollar en toda la organización y se pueden distribuir a las distintas instalaciones.
- **Crecimiento incremental.** En una instalación centralizada, un incremento de la carga de trabajo o la necesidad de un nuevo conjunto de aplicaciones normalmente supone la compra de gran equipamiento o de significativas actualizaciones de software. Esto implica un gasto importante. Además, estos cambios pueden requerir la conversión o reprogramación de aplicaciones existentes, con el correspondiente riesgo de errores y degradación del rendimiento. En un sistema distribuido, se pueden reemplazar gradualmente las aplicaciones o los sistemas, huyendo del enfoque de «todo o nada». Por último, los equipos antiguos se pueden dejar en las instalaciones para ejecutar aplicaciones sencillas, si el coste de pasar la aplicación a una nueva máquina no está justificado.
- **Mayor participación y control del usuario.** Con equipos más pequeños, más manejables, físicamente localizados cerca del cliente, el usuario tiene más posibilidades de influir en el diseño y funcionamiento del sistema, a través de la interacción con el personal técnico o a través de su supervisor inmediato.
- **Productividad del usuario final.** Los sistemas distribuidos tienden a tener mayor rapidez de respuesta, ya que cada pieza del equipo está realizando un trabajo más pequeño. Además, las aplicaciones y las interfaces pueden estar optimizadas a las necesidades del departamento. Los gestores del departamento pueden evaluar la efectividad de la parte local de la instalación y hacer los cambios apropiados.

Para lograr estos beneficios, el sistema operativo debe proporcionar diversas funciones de soporte para DDP. Estas incluyen el software para el intercambio de datos entre máquinas, la capacidad de los *clusters* de computadores de lograr alta disponibilidad y altas prestaciones y la capacidad de manejar procesos en entornos distribuidos.

En la era de la conectividad universal, de virus y de *hackers*, de escándalos y fraudes electrónicos, la seguridad se ha convertido en un aspecto fundamental. Dos tendencias hacen que este tema sea de particular interés. Primero, el crecimiento explosivo de los sistemas de computación y sus interconexiones a través de redes, ha incrementado la dependencia de las organizaciones y de los individuos en la información almacenada y en la comunicación entre los sistemas. Esto ha llevado a una gran concienciación sobre la necesidad de proteger los datos y recursos de su revelación, a garantizar la autenticidad de los datos y mensajes y a proteger los sistemas de ataques basados en la red. Segundo, las disciplinas de criptografía y seguridad han madurado, llevando al desarrollo de aplicaciones prácticas y de fácil disponibilidad para imponer la seguridad.

MAPA DE RUTA DE LA PARTE SEIS

CAPÍTULO 13. REDES

La comunicación de datos a través de la red y los sistemas distribuidos dependen del software de comunicaciones subyacente, que es independiente de las aplicaciones y que las descarga de gran parte de la carga del intercambio fiable de datos. Este software de comunicaciones está organizado en una arquitectura de protocolos, entre los que destacan los protocolos TCP/IP. El Capítulo 13 introduce el concepto de arquitectura de protocolos y proporciona una visión general de TCP/IP.

CAPÍTULO 14. PROCESAMIENTO DISTRIBUIDO, CLIENTE/SERVIDOR Y *CLUSTERS*

El Capítulo 14 examina el soporte que se require del sistema operativo para que cooperen múltiples sistemas. El capítulo analiza el cada vez más importante concepto de computación cliente/servidor, incluyendo la descripción de los dos mecanismos clave utilizados para implementar los sistemas cliente/servidor: paso de mensajes y llamadas a procedimiento remoto. El Capítulo 14 también examina el concepto de *clusters*.

CAPÍTULO 15. GESTIÓN DE PROCESOS DISTRIBUIDOS

El Capítulo 15 examina los aspectos fundamentales del desarrollo de sistemas operativos distribuidos. Primero, se analizan los requisitos y mecanismos de migración de procesos, que permiten que un proceso activo se mueva de una máquina a otra, con el objetivo de lograr un reparto equilibrado de carga y máxima disponibilidad. Luego se analiza el concepto de estado global distribuido, que es un elemento vital en el desarrollo de sistemas operativos distribuidos. Finalmente, se exploran dos aspectos clave en la gestión de procesos distribuidos: la exclusión mutua y el interbloqueo.

CAPÍTULO 16. SEGURIDAD

El Capítulo 16 proporciona una visión general de la seguridad en los computadores y sistemas operativos. El capítulo comienza con una visión general de las amenazas de seguridad. Se analizan los me-

canismos de protección de los sistemas de computación. A continuación, se describen las formas de contrarrestar las amenazas de intrusos: usuarios no autorizados o usuarios autorizados intentando realizar acciones no autorizadas. Para continuar, se analizan los virus, una de las amenazas mejor conocidas y más perjudiciales. El capítulo también analiza un exhaustivo mecanismo de seguridad conocido como el «sistema de confianza». Finalmente, se presenta la seguridad en redes.

CAPÍTULO 13

Redes

Junto con la creciente disponibilidad de computadores personales asequibles y potentes, ha habido una tendencia hacia el proceso de datos distribuido (DDP), en el que los procesadores, los datos y el procesamiento de datos pueden estar diseminados por toda la organización. Un sistema DDP implica dividir las funciones de computación y organizar de forma distribuida las bases de datos, el control de dispositivos y el control de interacciones.

En muchas organizaciones, existe una gran dependencia de los computadores personales y su asociación con los servidores. Los computadores personales se utilizan para dar soporte a aplicaciones de fácil manejo, tales como procesadores de texto, hojas de cálculo y presentación de gráficos. Los servidores mantienen las bases de datos corporativas y sofisticados gestores de bases de datos y software de sistemas de información. Se necesita una conexión entre los computadores personales y entre los computadores personales y los servidores. Existen varios mecanismos de uso común, desde tratar a los computadores personales como simples terminales, hasta tener un alto nivel de integración entre las aplicaciones de los computadores personales y los servidores.

Estas tendencias se han visto favorecidas por la evolución de las capacidades de los sistemas operativos y por las utilidades de soporte. Se han explorado diversas soluciones distribuidas:

- **Arquitectura de comunicaciones.** Es un software que da soporte a un grupo de computadores en red. Proporciona soporte para aplicaciones distribuidas, tales como correo electrónico, transferencia de ficheros y acceso a terminales remotos. Sin embargo, los computadores siguen siendo entidades independientes para los usuarios y para las aplicaciones, que se deben comunicar entre sí por expreso deseo. Cada computador tiene su propio sistema operativo y es posible tener una mezcla de computadores y sistemas operativos, siempre y cuando todas las máquinas soporten la misma arquitectura de comunicaciones. La arquitectura de comunicaciones más ampliamente utilizada es el conjunto de protocolos TCP/IP, que se examina en este capítulo.

- **Sistema operativo de red.** En esta configuración hay una red de máquinas, normalmente estaciones de trabajo de un solo usuario, y una o más máquinas servidoras. Éstas proporcionan servicios de red o aplicaciones, tales como almacenamiento de ficheros y gestión de impresión. Cada computador tiene su propio sistema operativo. El sistema operativo de red es un añadido al sistema operativo local, que permite a las máquinas interactuar con los servidores. El usuario conoce la existencia de múltiples computadores y debe trabajar con ellos de forma explícita. Normalmente se utiliza una arquitectura de comunicaciones común para dar soporte a estas aplicaciones de red.

- **Sistema operativo distribuido.** Es un sistema operativo común compartido por una red de computadores. A los usuarios les parece un sistema operativo normal centralizado, pero les proporciona acceso transparente a los recursos de diversas máquinas. Un sistema operativo distribuido puede depender de una arquitectura de comunicaciones para las funciones básicas de comunicación, pero normalmente se incorporan un conjunto de funciones de comunicación más sencillas para proporcionar mayor eficiencia.

La tecnología de la arquitectura de comunicaciones está bien desarrollada y es soportada por todos los vendedores. Los sistemas operativos de red son un fenómeno más reciente, pero existen algunos productos comerciales. La parte central de investigación y desarrollo de sistemas distribuidos se centra en el área de sistemas operativos distribuidos.

Aunque ya se han introducido algunos sistemas comerciales, los sistemas operativos distribuidos complementamente funcionales están todavía en etapa experimental. En este capítulo y los dos siguientes, se examinan las capacidades del procesamiento distribuido. Este capítulo se centra en el software de protocolos de red subyacente.

13.1. LA NECESIDAD DE UNA ARQUITECTURA DE PROTOCOLOS

Cuando un computador, terminal y/u otro dispositivo de procesamiento de datos intercambia datos, el procedimiento involucrado puede ser muy complejo. Considere, por ejemplo, la transferencia de un fichero entre dos computadores. Debe haber una ruta de datos entre los computadores, ya sea un enlace directo o una red de comunicaciones. Pero se necesita más. Entre las típicas tareas que se deben realizar se incluyen las siguientes:

1. El sistema emisor debe activar el enlace directo de comunicación de datos o debe informar a la red de comunicaciones de la identidad del sistema destinatario deseado.
2. El sistema emisor debe verificar que el sistema de destino está preparado para recibir datos.
3. La aplicación de transferencia de ficheros del sistema origen debe verificar que el programa de gestión de ficheros del sistema destino está preparado para aceptar y almacenar el fichero de ese usuario particular.
4. Si los formatos de los ficheros o las representaciones de datos en los sistemas son incompatibles, uno de los dos sistemas deberá ejecutar una función de traducción de formato.

El intercambio de información entre computadores con finalidad cooperativa se conoce como *comunicación de computadores*. De forma similar, cuando uno o más computadores se interconectan a través de una red de comunicación, el conjunto de computadores se denomina *red de computadores*. Ya que se requiere un nivel similar de cooperación entre un terminal y un computador, estos términos también se utilizan cuando alguna de las entidades de comunicación son terminales.

En relación a la comunicación de computadores y redes de computadores, hay dos conceptos de suma importancia:

- Protocolos.
- Arquitectura de comunicaciones o arquitectura de protocolos.

Un **protocolo** se utiliza para comunicar entidades de diferentes sistemas. Los términos *entidad* y *sistemas* se utilizan en un sentido muy genérico. Algunos ejemplos de entidades son los programas de aplicación de usuario, paquetes de transferencia de ficheros, sistemas de gestión de bases de datos, servicios de correo electrónico y terminales. Algunos ejemplos de sistemas son computadores, terminales y sensores remotos. Fijarse que en algunos casos la entidad y el sistema en que reside son los mismos (por ejemplo, terminales). En general, una entidad es cualquier cosa capaz de enviar y recibir información, y un sistema es un objeto físico que contiene una o más entidades. Para que dos entidades se comuniquen con éxito, deben «hablar el mismo idioma». Lo que se comunica, cómo se comunica y cuándo se comunica, debe hacerse de acuerdo a unas convenciones entre las entidades involucradas. Las convenciones se denominan protocolos, que se pueden definir como un conjunto de reglas que gobiernan el intercambio de datos entre dos entidades. Los elementos principales de un protocolo son los siguientes:

- **Sintaxis.** Incluye cosas tales como formatos de datos y niveles de señales.
- **Semántica.** Incluye información de control para realizar coordinación y gestión de errores.
- **Temporización.** Incluye ajuste de velocidades y secuenciamiento.

El Apéndice 13A contiene un ejemplo específico de un protocolo, el estándar de Internet Protocolo de Transferencia Simple de Ficheros (*Trivial File Transfer Protocol*, TFTP).

Conociendo el concepto de protocolo, se puede introducir el concepto de **arquitectura de protocolos**. Está claro que debe existir un alto nivel de cooperación entre los dos sistemas de computación. En lugar de implementar esta lógica con un solo módulo, la tarea se puede descomponer en subtareas, cada una de las cuales se puede implementar de forma individual. Por ejemplo, la Figura 13.1 sugiere la forma en que se podría implementar un servicio de transferencia de ficheros utilizando tres módulos. Las tareas 3 y 4 de la lista anterior se podrían realizar en un módulo de transferencia de ficheros. Los dos módulos de los dos sistemas intercambian ficheros y mandatos. Sin embargo, el módulo de transferencia de ficheros no se encarga de los detalles de transmitir los datos y mandatos, confía esta misión a un módulo de servicios de comunicaciones. Este módulo es responsable de asegurarse de que los mandatos de transferencia de ficheros y los datos se intercambian de forma fiable entre los dos sistemas. Posteriormente, se explorará la forma en que trabaja un módulo de servicios de comunicaciones que, entre otras cosas, realiza la tarea 2. Por último, la naturaleza del intercambio entre los dos módulos de servicio de comunicaciones es independiente de la naturaleza de la red que los interconecta. Por tanto, mejor que construir los detalles de la interfaz de red en el módulo de servicios de comunicaciones, tiene sentido tener un tercer módulo, un módulo de acceso a red, que realiza la tarea 1.

Para resumir, el módulo de transferencia de ficheros contiene toda la lógica que es única a la aplicación de transferencia de ficheros, tal como la transmisión de claves, mandatos y registros de los ficheros. Estos ficheros y mandatos se deben transmitir de forma fiable. Sin embargo, estos requisitos de fiabilidad son comunes a más aplicaciones (por ejemplo, correo electrónico, transferencia de documentos). Por tanto, estos requisitos se acometen en el módulo de servicios de comunicaciones que se puede utilizar por múltiples aplicaciones. El módulo de servicios de comunicaciones se preocupa de que los dos sistemas de computación estén activos y listos para el intercambio de datos y de llevar la cuenta de los datos que se están intercambiando para asegurar su entrega. Sin embargo, estas tareas son independientes del tipo de red que se está utilizando. Por tanto, la lógica para tratar con la red, se pone en el módulo de acceso a red. Si cambia la red en uso, sólo se ve afectado este módulo.

De esta manera, en lugar de tener un único módulo para realizar las comunicaciones, hay un conjunto estructurado de módulos que implementan la función de comunicaciones. Esta estructura se conoce como una arquitectura de protocolos. En este punto puede ser útil una analogía. Suponga un ejecutivo en la oficina X que quiere enviar un documento a un ejecutivo en la oficina Y. El ejecutivo en X prepara el documento y, a lo mejor, le añade una nota. Esto se corresponde con las acciones de la aplicación de transferencia de ficheros en la Figura 13.1. A continuación, el ejecutivo en X entrega el

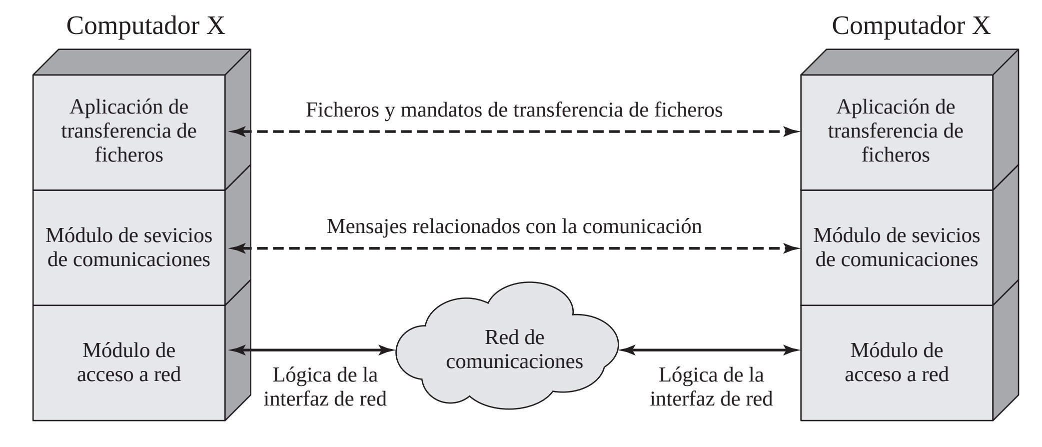


Figura 13.1. Una arquitectura simplicada de transferencia de ficheros.

documento a un secretario o administrativo (A). A en X pone el documento en un sobre y escribe la dirección de Y y la dirección del remitente X. Puede ser que el sobre esté marcado como «confidencial». Las acciones de A se corresponden con el módulo de servicios de comunicaciones de la Figura 13.1. A en X entrega el paquete a un departamento de envíos. Alguien en el departamento de envíos decide cómo enviar el paquete: correo, correo urgente, etc. El departamento de envíos añade el franqueo adecuado al paquete y lo envía. El departamento de envíos se corresponde con el módulo de acceso a red de la Figura 13.1. Cuando el paquete llega a Y, ocurren una serie de acciones similares. El departamento de envíos de Y recibe el paquete que le entrega a A o secretario correspondiente, basándose en el nombre del paquete. A abre el paquete y entrega el documento adjunto al ejecutivo al que va dirigido.

13.2. LA ARQUITECTURA DE PROTOCOLOS TCP/IP

TCP/IP es el resultado de la investigación y el desarrollo sobre protocolos llevado a cabo en la red de intercambio de paquetes, ARPANET, financiada por el *Defense Advanced Research Projects Agency* (DARPA). Normalmente se le conoce como paquete de protocolos TCP/IP. Este paquete de protocolos tiene una colección de protocolos que se han utilizado como los estándares de Internet por el *Internet Activities Board* (IAB). En el sitio web de este libro hay un documento que proporciona una discusión sobre los estándares de Internet.

CAPAS TCP/IP

En términos generales, las comunicaciones involucran tres agentes: aplicaciones, computadores y redes. La transferencia de ficheros o el correo electrónico son algunos ejemplos de aplicación. Las aplicaciones de interés en este capítulo son las aplicaciones distribuidas que implican intercambio de datos entre dos sistemas de computación. Estas aplicaciones, y otras, ejecutan sobre computadores que a menudo pueden tener múltiples aplicaciones al mismo tiempo. Los computadores están conectados a redes y los datos que se intercambian se transfieren por la red de un computador a otro. De esta forma, la transferencia de datos de una aplicación a otra implica obtener los datos del computador en el que reside la aplicación y además obtener los datos de la aplicación deseada.

No existe un modelo oficial de protocolos TCP/IP. Sin embargo, basándose en los protocolos estándares que han sido desarrollados, las tareas de comunicación del TCP/IP se pueden organizar en cinco capas relativamente independientes. De abajo a arriba:

- Capa física.
- Capa de acceso a red.
- Capa de Internet.
- Máquina-a-máquina, o capa de transporte.
- Capa de aplicación.

La **capa física** cubre la interfaz física entre un dispositivo de transmisión de datos (por ejemplo, una estación de trabajo o un computador) y un medio de transmisión o red. Esta capa se preocupa de las características del medio de transmisión, la naturaleza de las señales, la tasa de datos y aspectos similares.

La **capa de acceso a red** se preocupa del intercambio de datos entre un sistema final (servidor, estación de trabajo, etc.) y la red a la que está unido. El computador que realiza el envío debe

proporcionar a la red la dirección del computador destino, para que la red pueda encaminar los datos al destino apropiado. El computador emisor podría desear utilizar ciertos servicios, tales como la prioridad, que pueden ser proporcionados por la red. El software específico utilizado en esta capa depende del tipo de red a ser utilizado; se han desarrollado diferentes estándares para conmutación de circuitos, conmutación de paquetes (por ejemplo, *frame relay*), LANs (por ejemplo, *Ethernet*), y otras. De esta forma, se hace posible separar estas funciones, relacionadas con la red, en una capa separada. Gracias a ello, el restante software de comunicaciones no se tiene que preocupar de los aspectos específicos de la red a utilizar, funcionando correctamente con independiencia de la red.

La capa de acceso a red se preocupa de enviar datos a través de una red para dos sistemas finales unidos a la misma. En los casos en que los dispositivos estén unidos a distintas redes, se necesita un procedimiento para permitir a los datos atravesar múltiples redes interconectadas. Esta es la función de la capa de Internet. El **Protocolo de Internet** (*Internet Protocol*, IP) se utiliza en esta capa para proporcionar la función de encaminamiento a través de múltiples redes. Este protocolo no sólo está implementado en sistemas finales, también está implementado en los encaminadores (*routers*). Un encaminador es un procesador que conecta dos redes y cuya función principal es pasar datos de una red a la otra, en la ruta desde el sistema final origen hasta el sistema final destino.

Independientemente de la naturaleza de las aplicaciones que están intercambiando los datos, suele existir el requisito de la fiabilidad. Es decir, se pretende asegurar que los datos lleguen a la aplicación de destino y que lo hagan en el mismo orden en que fueron enviados. Como se verá, los mecanismos para proporcionar fiabilidad son independientes de la naturaleza de las aplicaciones. De esta forma, tiene sentido unir estos mecanimos en una capa común compartida por todas las aplicaciones; esta es la capa máquina-a-máquina o **capa de transporte**. El Protocolo de Control de Transmisión (*Transmission Control Protocol*, TCP) es el protocolo de uso más frecuente utilizado para proporcionar esta funcionalidad.

Finalmente, la **capa de aplicación** contiene la lógica necesaria para soportar las aplicaciones de usuario. Para cada tipo diferente de aplicación, tal como una aplicación de transferencia de ficheros, se necesita un módulo diferente.

TCP Y UDP

Para la mayor parte de las aplicaciones que utilizan la arquitectura de protocolos TCP/IP, el protocolo de la capa de transporte es TCP. TCP proporciona una conexión fiable para la transmisión de datos entre dos aplicaciones. Una conexión es simplemente una asociación lógica temporal entre dos entidades de diferentes sistemas. Durante la duración de la conexión, cada entidad hace un seguimiento de los segmentos, para poder regular el flujo de los mismos y recuperar segmentos perdidos o dañados.

La Figura 13.2a muestra el formato de una cabecera TCP, que tiene un mínimo de 20 octetos ó 160 bits. Los campos Puerto Origen y Puerto Destino identifican a las aplicaciones en los sistemas origen y destino de esta conexión. Los campos Número de Secuencia, Número de Confirmación y Ventana proporcionan control de flujo y control de errores. La Suma de Control (*checksum*) es un código de 16 bits basado en el contenido del segmento y que se utiliza para detectar errores.

Además de TCP, hay otro protocolo en la capa de transporte que se usa bastante: el Protocolo de Datagramas de Usuario (*User Datagram Protocol*, UDP). UDP no garantiza la entrega, que se preserve la secuencia o la protección frente a la duplicación. UDP permite a un proceso enviar un mensaje a otro proceso con los mínimos mecanismos de protocolo posibles. Algunas aplicaciones orientadas a transacciones utilizan UDP; un ejemplo es SNMP (*Simple Network Management Protocol*), el proto-

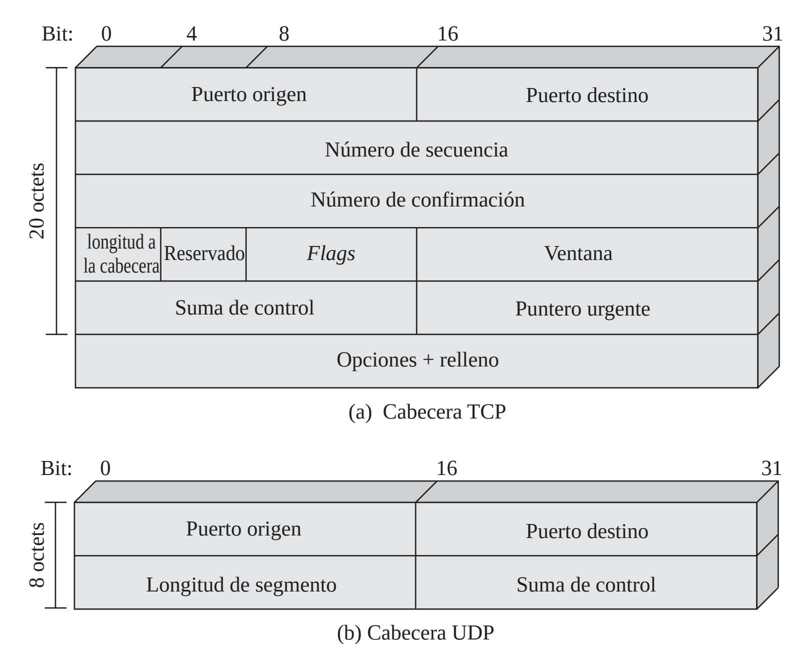


Figura 13.2. Cabeceras TCP y UDP.

colo estándar de gestión de redes TCP/IP. Ya que no es orientado a conexión, UDP no tiene mucho trabajo. Esencialmente, añade el servicio del uso de puertos a IP. Esto se ve mejor examinando la cabecera UDP, mostrada en la Figura 13.2b.

IP Y IPV6

Durante décadas, IP ha sido la pieza clave de la arquitectura de protocolos TCP/IP. La Figura 13.3a muestra el formato de cabecera IP, que tiene un mínimo de 20 octetos ó 160 bits. La cabecera, junto con el segmento de la capa de transporte, forma un bloque de nivel IP conocido como **datagrama IP** o *paquete IP*. La cabecera incluye direcciones de origen y de destino de 32 bits. El campo Suma de Control de la cabecera se utiliza para detectar errores en la cabecera con el propósito de evitar fallos de envío. El campo Protocolo indica qué protocolo de capa superior está utilizando IP (TCP, UDP o algún otro). Los campos ID, *Flags* y Desplazamiento de Fragmento (*Fragment Offset*) se utilizan en los procesos de fragmentación y reensamblado, en los que un datagrama IP se divide en múltiples datagramas IP durante la transmisión y se vuelven a unir en el destino.

En 1995, el *Internet Engineering Task Force* (IETF), que desarrolla estándares de protocolos para Internet, publicó la especificación para la siguiente generación de IP, conocida como IPng. Esta especificación se transformó en estándar en 1996 con el nombre IPv6. IPv6 proporciona una serie de mejoras funcionales sobre IP, y fue diseñado para ajustarse a las redes de mayor velocidad existentes hoy en día y para la mezcla de *streams* de datos, incluyendo gráficos y vídeo, que están siendo cada vez más comunes. Pero la verdadera razón para desarrollar el nuevo protocolo fue la necesidad de más direccionamiento. El IP actual utiliza direcciones de 32 bits para especificar la fuente y el destino. Con el crecimiento explosivo de Internet y de las redes privadas unidas a Internet, esta longitud de direccionamiento se hace insuficiente para acomodar a todos los sistemas. Como muestra la Figura 13.3b, IPv6 incluye campos de dirección de 128 bits.

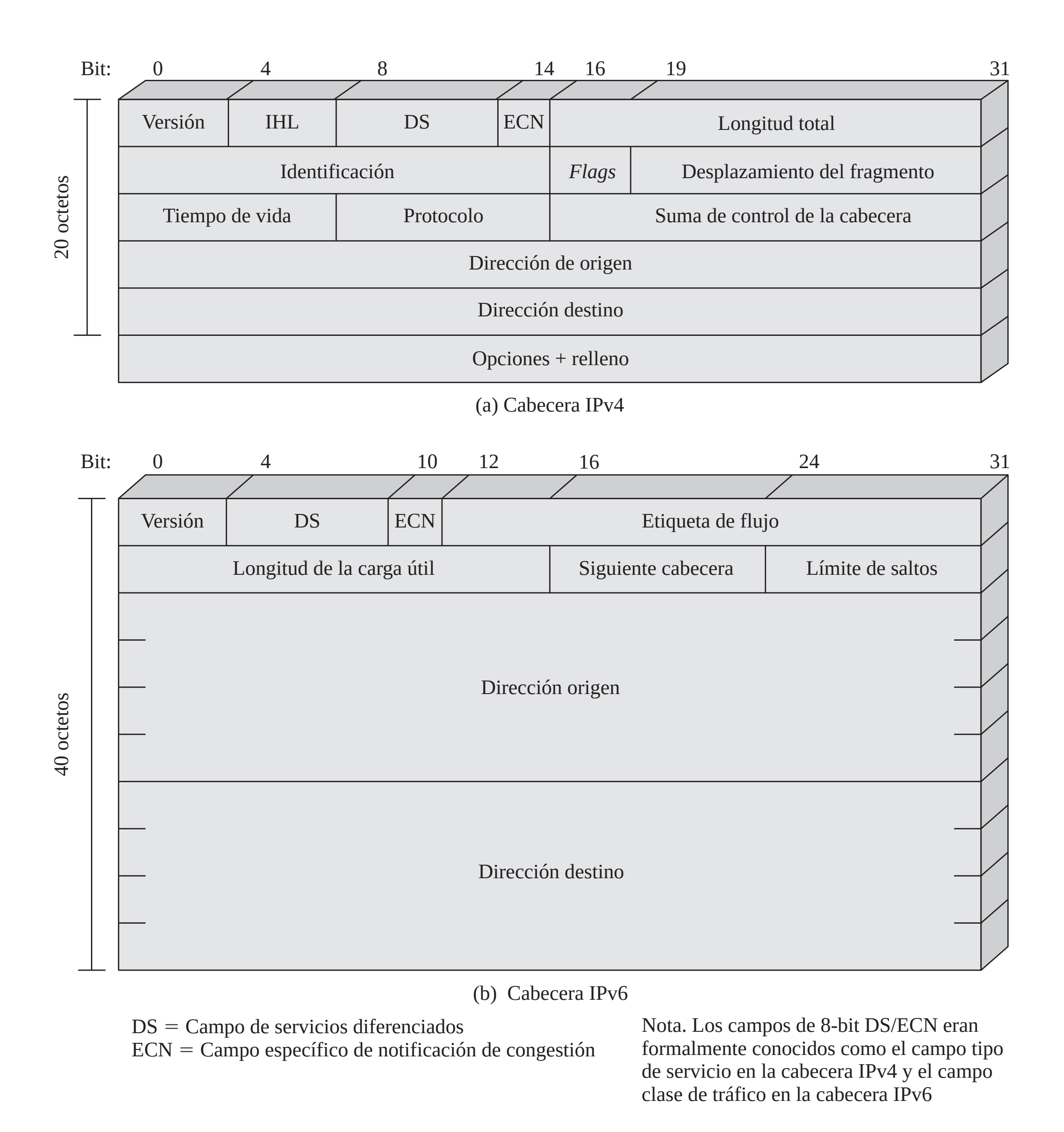


Figura 13.3. Cabeceras IP.

Todas las instalaciones que estén utilizando TCP/IP se espera que migren del actual IP a IPv6, pero este proceso llevará muchos años, o décadas.

FUNCIONAMIENTO DE TCP/IP

La Figura 13.4 muestra cómo se configuran estos protocolos para realizar la comunicación. A la hora de conectar un computador a la red se utiliza algún tipo de protocolo de acceso a red, tal como Ethernet. Este protocolo permite a una máquina enviar datos a través de la red a otra máquina o, en caso de que la máquina esté en otra red, al encaminador. IP está implementado en todos los sistemas finales y

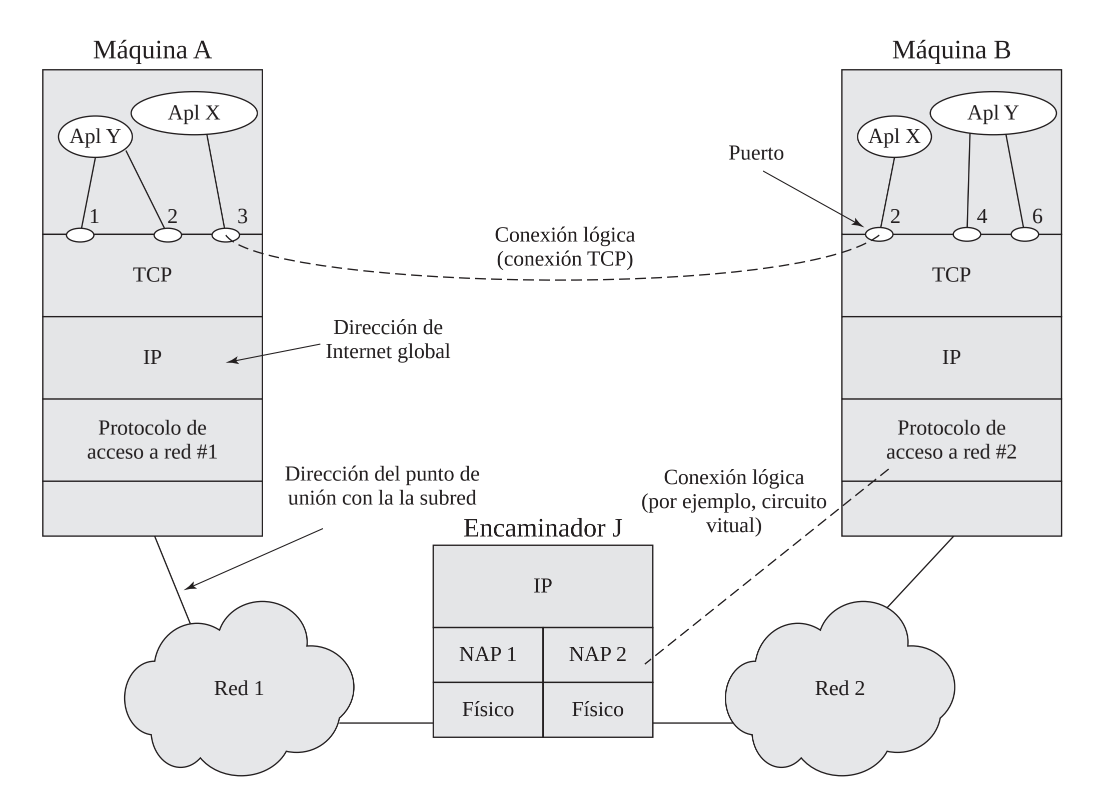


Figura 13.4. Conceptos TCP/IP.

encaminadores. TCP está implementado solamente en los sistemas finales; lleva control de los bloques de datos que están siendo transferidos para asegurar que todos se envían de forma fiable a la aplicación apropiada.

Para poder realizar una comunicación satisfactoria, toda entidad del sistema debe tener una dirección única. De hecho, se necesitan dos niveles de direcciones. Cada máquina en una red debe tener una direccion Internet global única; esto permite que los datos se entreguen en la máquina apropiada. Esta dirección es la utilizada por IP para la entrega. Cada aplicación de una máquina debe tener una dirección única en la máquina; esto permite a TCP enviar los datos al proceso adecuado. Estas últimas direcciones se conocen como puertos.

Hagamos el seguimiento de una operación sencilla. Suponga que un proceso, asociado con el puerto 3 en la máquina A, quiere enviar un mensaje a otro proceso, asociado con el puerto 2 de la máquina B. El proceso en A manda un mensaje a TCP con instrucciones de enviarlo a la máquina B, puerto 2. TCP manda el mensaje a IP con instrucciones de enviarlo a la máquina B. Es importante observar que no se necesita decir a IP la identificación del puerto de destino, ya que todo lo que necesita saber es que los datos son para la máquina B. A continuación, IP manda el mensaje a la capa de acceso a red (por ejemplo, Ethernet) con instrucciones de mandarlo al encaminador J (el primer salto en el camino a B).

Para controlar esta operación, se debe transmitir tanto información de control como información de usuario, como se sugiere en la Figura 13.5. El proceso emisor genera un bloque de datos que pasa a TCP. TCP puede romper este bloque en piezas más pequeñas para hacerlo más manejable. TCP añade información de control, conocida como cabecera TCP (Figura 13.2a), a cada una de estas piezas,

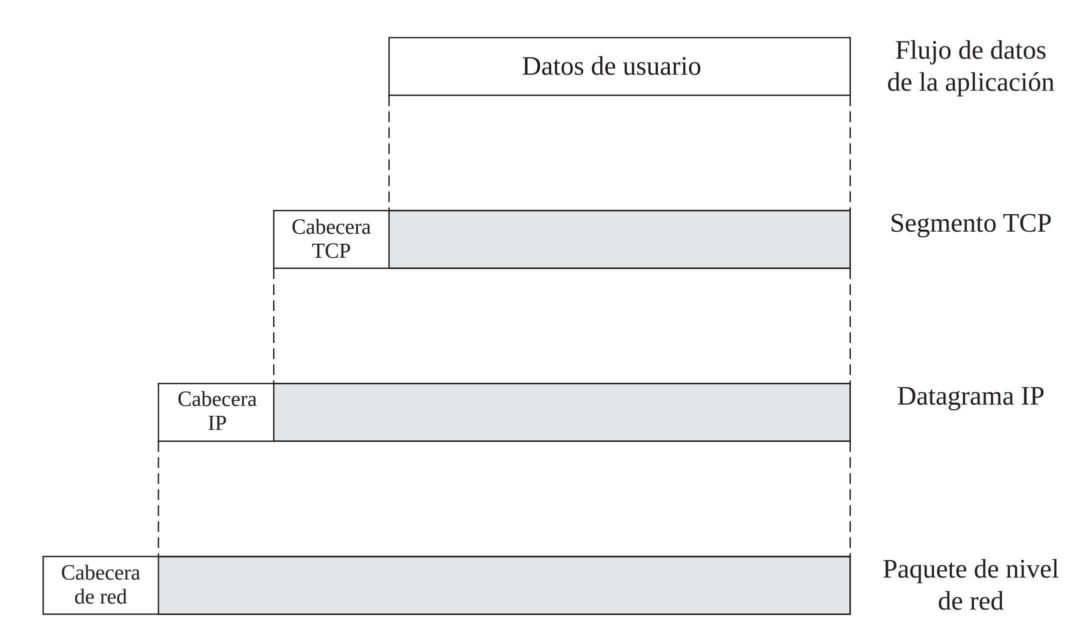


Figura 13.5. Unidades de Datos de Protocolo (PDUs) en la arquitectura TCP/IP.

formando un **segmento TCP**. La información de control será utilizada por el protocolo TCP correspondiente de la entidad en la máquina B.

A continuación, TCP pasa cada segmento a IP, con instrucciones de transmitirlo a B. Estos segmentos se deben transmitir a través de una o más redes y deben pasar por uno o más encaminadores intermedios. Esta operación, además, requiere del uso de información de control. De esta forma, IP añade una cabecera con información de control (Figura 13.3) a cada segmento para formar un datagrama IP. Un ejemplo de un elemento de la cabecera IP es la dirección de la máquina de destino (en este ejemplo, B).

Por último, cada datagrama IP se manda a la capa de acceso a red para su transmisión a través de la primera red de su viaje al destino. La capa de acceso a red añade su propia cabecera, creando un paquete o trama. El paquete se transmite a través de la red al encaminador J. La cabecera del paquete contiene la información que la red necesita para transmitir los datos. Ejemplos de elementos que pueden estar en esta cabecera son:

- **Dirección de red del destino.** La red necesita saber a qué dispositivo entregar el paquete, en este caso al encaminador J.
- **Petición de servicios.** El protocolo de acceso a red podría solicitar el uso de algunos servicios de red, como la prioridad.

En el encaminador J, se quita la cabecera del paquete y se examina la cabecera IP. Basándose en la información de la dirección de destino en la cabecera IP, el módulo IP del encaminador dirige el datagrama a través de la red 2 a B. Para hacer esto, se añade de nuevo al datagrama una cabecera de acceso a red.

Cuando se reciben los datos en B, sucede el proceso inverso. En cada capa, se quita la correspondiente cabecera y el resto se pasa a la capa superior, hasta que los datos originales del usuario se entregan en el proceso destino.

APLICACIONES TCP/IP

Existen diversas aplicaciones estandarizadas para su uso con TCP. A continuación se comentan tres de las más comunes.

El **Protocolo Simple de Transferencia de Correo** (*Simple Mail Transfer Protocol*, SMTP) proporciona un servicio básico de correo electrónico. SMTP proporciona un mecanismo para transmitir mensajes entre máquinas. Algunas características de SMTP son: listas de correo, confirmaciones y reenvío. El protocolo SMTP no especifica la forma en que se deben crear los mensajes; se requiere algún servicio de edición de correo electrónico. Una vez que el mensaje está creado, SMTP lo acepta y utiliza TCP para su envío a un módulo SMTP de otra máquina. El módulo SMTP destino hará uso de un paquete de correo electrónico local para almacenar el mensaje entrante en el buzón del usuario.

El **Protocolo de Transferencia de Ficheros** (*File Transfer Protocol*, FTP) se utiliza para enviar ficheros de un sistema a otro por petición del usuario. Se pueden enviar tanto ficheros de texto como binarios. El protocolo proporciona características para el control de acceso de los usuarios. Cuando un usuario quiere realizar una transferencia de fichero, FTP establece una conexión TCP con el sistema destino para el intercambio de mensajes de control. Esta conexión permite enviar identificadores de usuario y claves, y le permite al usuario especificar el fichero y las acciones deseadas. Una vez que se ha aprobado una transferencia de ficheros, se establece una segunda conexión TCP para el envío de los datos. El fichero se transmite sobre la conexión, sin la sobrecarga de ninguna cabecera o información de control en el nivel de aplicación. Cuando se completa la transferencia, se usa la conexión de control para indicar la finalización y para aceptar nuevos mandatos de transferencia de ficheros.

TELNET proporciona un servicio de inicio de sesión remoto, que le permite a un usuario en un terminal o computador personal iniciar una sesión en un computador remoto y trabajar como si estuviera conectado a este computador. El protocolo se diseñó para trabajar con terminales con modo de *scroll* sencillo. En la actualidad, TELNET está implementado en dos módulos: *TELNET Usuario* interactúa con el módulo de E/S del terminal para comunicarse con un terminal local. Convierte las características del terminal al estándar de red y viceversa. *TELNET Servidor* interactúa con la aplicación, actuando como un manejador de terminal sucedáneo, de forma que los terminales remotos le parecen locales a la aplicación. El tráfico de terminal entre TELNET Usuario y Servidor se realiza a través de una conexión TCP.

13.3. *SOCKETS*

El concepto de *socket* y de programación con *sockets* se desarrolló en los años ochenta en el entorno UNIX como la Interfaz de *Sockets* de Berkeley (*Berkeley Sockets Interface*). En esencia, un *socket* permite la comunicación entre un proceso cliente y un proceso servidor y puede ser orientado a conexión o no orientado a conexión. Un *socket* se puede considerar como un punto final en una comunicación. Un socket de cliente en un computador utiliza una dirección para llamar a un *socket* de servidor en otro computador. Una vez que han entrado en comunicación los *sockets*, los dos computadores pueden intercambiar datos.

Normalmente, los computadores con servidores basados en *sockets* mantienen abierto un puerto TCP o UDP, preparado para llamadas entrantes no planificadas. El cliente suele determinar la identificación del *socket* del servidor deseado buscando en una base de datos de Sistema de Nombres de Dominio (*Domain Name System*, DNS). Una vez que se realiza la conexión, el servidor redirije el diálogo a un puerto diferente para liberar el número de puerto principal para nuevas llamadas entrantes.

Aplicaciones de Internet, tales como TELNET o *rlogin* (inicio de sesión remoto), hacen uso de los *sockets*, ocultando los detalles al usuario. Sin embargo, se pueden utilizar los *sockets* en un programa (en lenguajes como C o Java), permitiendo al programador soportar de forma fácil funciones y aplicaciones de red. El mecanismo de programación de *sockets* incluye semántica suficiente para permitir que dos procesos no relacionados de dos máquinas se puedan comunicar.

La *Berkeley Sockets Interface* es el API (Interfaz de Programación de Aplicaciones) estándar para el desarrollo de aplicaciones de red, en un amplio rango de sistemas operativos. *Windows Sockets* (WinSock) está basado en la especificación de Berkeley. El API de los *sockets* proporciona acceso genérico a servicios de comunicación entre procesos. De esta forma, los *sockets* son ideales para que los estudiantes aprendan los principios de los protocolos y de las aplicaciones distribuidas a través del desarrollo del programas.

EL *SOCKET*

Recordar que cada cabecera TCP y UDP incluye los campos del puerto origen y puerto destino (Figura 13.2). Estos valores de **puerto** identifican a los respectivos usuarios (aplicaciones) de las dos entidades TCP. Además, cada cabecera IPv4 e IPv6 incluye los campos con las direcciones origen y destino (Figura 13.3); estas **direcciones IP** identifican a los respectivos sistemas. La concatenación de un valor de puerto y una dirección IP forma un *socket*, que es único en Internet. De esta forma, en la Figura 13.4, la combinación de las direcciones IP de la máquina B y el número de puerto de la aplicación X identifican de forma única la localización del *socket* de la aplicación X en la máquina B. Como indica la figura, una aplicación puede tener múltiples direcciones de *sockets*, una por cada puerto de la aplicación.

El *socket* se utiliza para definir una **interfaz de programación de aplicaciones** (API), que es una interfaz genérica de comunicaciones para escribir programas que usan TCP y UDP. En la práctica, cuando se usa como una API, un *socket* se identifica por un triplete (protocolo, dirección local, proceso local). La dirección local es una dirección IP y el proceso local es un número de puerto. Ya que los número de los puertos son únicos en un sistema, el número del puerto implica el protocolo (TCP o UDP). Sin embargo, por claridad y facilidad de implementación, los *sockets* utilizados en el API incluyen un protocolo así como la dirección IP y el número de puerto para definir un *socket*.

Correspondiéndose con los dos protocolos, el API de *Sockets* reconoce dos tipos de *sockets*: *sockets stream* y *sockets* datagrama. Los **socket *stream*** hacen uso de TCP y proporcionan una transferencia de datos fiable orientada a conexión. Por tanto, con *sockets stream*, se garantiza que todos los bloques de datos enviados entre un par de sockets llegan en el orden en que se enviaron. Los ***sockets datagrama***, hacen uso de UDP, que no proporciona las características orientadas a conexión de TCP. Por tanto, con *sockets datagrama*, ni se garantiza la entrega, ni que se mantenga el orden.

Hay un tercer tipo de *sockets* proporcionados por el API de *Sockets*: *sockets raw*. Los sockets *raw* permiten acceso directo a las capas más bajas del protocolo, tales como IP.

LLAMADAS DE LA INTERFAZ *SOCKET*

Esta subsección resume las principales llamadas al sistema.

Configuración del *socket* El primer paso en la utilización de un *Socket* es crear el nuevo *socket* utilizando el mandato `socket().` Este mandato incluye tres parámetros; la familia del protocolo, que siempre es `PF_INET` para usar los protocolos TCP/IP. *Tipo*, que especifica si es un *socket stream* o datagrama, y *protocolo*, que especifica TCP o UDP. La razón de tener que especificar *tipo* y *proto-*

colo es para permitir nuevos protocolos de nivel de transporte en futuras implementaciones. De esta forma, podría haber más de un protocolo de transporte de tipo datagrama y más de un protocolo de transporte orientado a conexión. El comando `socket()` devuelve un valor entero que identifica este *socket*; es similar a un descriptor de fichero UNIX. La estructura de datos de los *sockets* depende de la implementación. Incluye el puerto y la dirección IP origen y, si hay una conexión abierta o pendiente, el puerto y la dirección IP destino y varias opciones y parámetros asociados con la conexión.

Después de que se cree un *socket* se le debe asignar una dirección en la que escuchar. La función `bind()` enlaza un *socket* con una dirección *socket*. La dirección tiene la estructura:

```
struct sockaddr_in {
    short int sin_family;          // Familia de direcciones (TCP/IP)
    unsigned short int sin_port;   // Número de puerto
    struct in_addr sin_addr;       // Dirección de Internet
    unsigned char sin_zero[8];     // Mismo tamaño que struct sockaddr
};
```

Conexión del *socket*

Para un *socket stream*, una vez creado, se debe establecer una conexión con el *socket* remoto. Una parte funciona como un cliente, y solicita una conexión a la otra parte, que actúa como servidor.

La configuración de la parte servidora de una conexión requiere dos pasos. Primero, la aplicación servidora realiza un `listen()`, para indicar que un determinado *socket* está listo para aceptar conexiones entrantes. El parámetro *backlog* es el número de conexiones permitidas en la cola de entrada. Cada una de las conexiones que llega se sitúa en esta cola hasta que el servidor realiza un `accept()` para ella. Por tanto, `accept()` se utiliza para extraer una solicitud de la cola. Si la cola está vacia, la llamada `accept()` bloquea al proceso hasta que llega una petición de conexión. Si hay una llamada esperando, el `accept()` devuelve un nuevo descriptor de fichero para la conexión. Esto crea un nuevo *socket*, que tiene la dirección IP y el número de puerto de la parte remota, la dirección IP de este sistema y un nuevo número de puerto. La razón de que se asigne un nuevo *socket* y un nuevo número de puerto es para permitir a la aplicación local seguir escuchando más peticiones. Como resultado, una aplicación podría tener múltiples conexiones activas al mismo tiempo, cada una con diferente número de puerto local. Este nuevo número de puerto se le devuelve al sistema que realizó la petición a través de la conexión TCP.

Una aplicación cliente realiza un `connect()`, que especifica tanto el *socket* local como la dirección del *socket* remoto. Si el intento de conexión no es satisfactorio, la llamada devuelve un –1. Si el intento es satisfactorio, la llamada devuelvo un 0 y rellena el parámetro de descripción de la conexión para incluir la dirección IP y el número de puerto de los *sockets* local y remoto. Recuerde que el número de puerto remoto puede ser diferente del especificado en el parámetro **foreignAddress**, porque el número de puerto se cambia en la máquina remota.

Una vez que la conexión está establecida, se puede utilizar **getpeername()** para averiguar quién está en el otro extremo del *socket*. Esta función devuelve un valor en el parámetro **sockfd**.

Comunicación *Socket*. En una conexión *stream*, se utilizan las llamadas `send()` y `recv()` para enviar y recibir datos sobre la conexión identificada por el parámetro `sockfd`. En la llamada `send()`, el parámetro `*msg` apunta al bloque de datos a ser enviado y el parámetro len especifica el número de bytes a enviar. El parámetro `flags` contiene *flags* de control, y normalmente se deja con el valor 0. La llamada `send()` devuelve el número de bytes enviados, que puede ser menor que el

número especificado en el parámetro `len`. En la llamada `recv()`, el parámetro `*buf` apunta al *buffer* para almacenar los datos entrantes, con un número máximo de bytes establecido con el parámetro `len`.

En cualquier momento, cualquiera de las partes puede cerrar la conexión con la llamada `close()`, que evita más envíos y recepciones. La llamada `shutdown()` permite al llamante dejar de enviar, de recibir, o ambos.

La Figura 13.6 muestra la interacción de las partes cliente y servidora en el establecimiento, uso y finalización de una conexión.

Para **comunicaciones datagrama**, se utilizan las funciones `sendto()` y `recvfrom()`. La llamada `sendto()` incluye todos los parámetros de la llamada `send()`, pero además especifica la di-

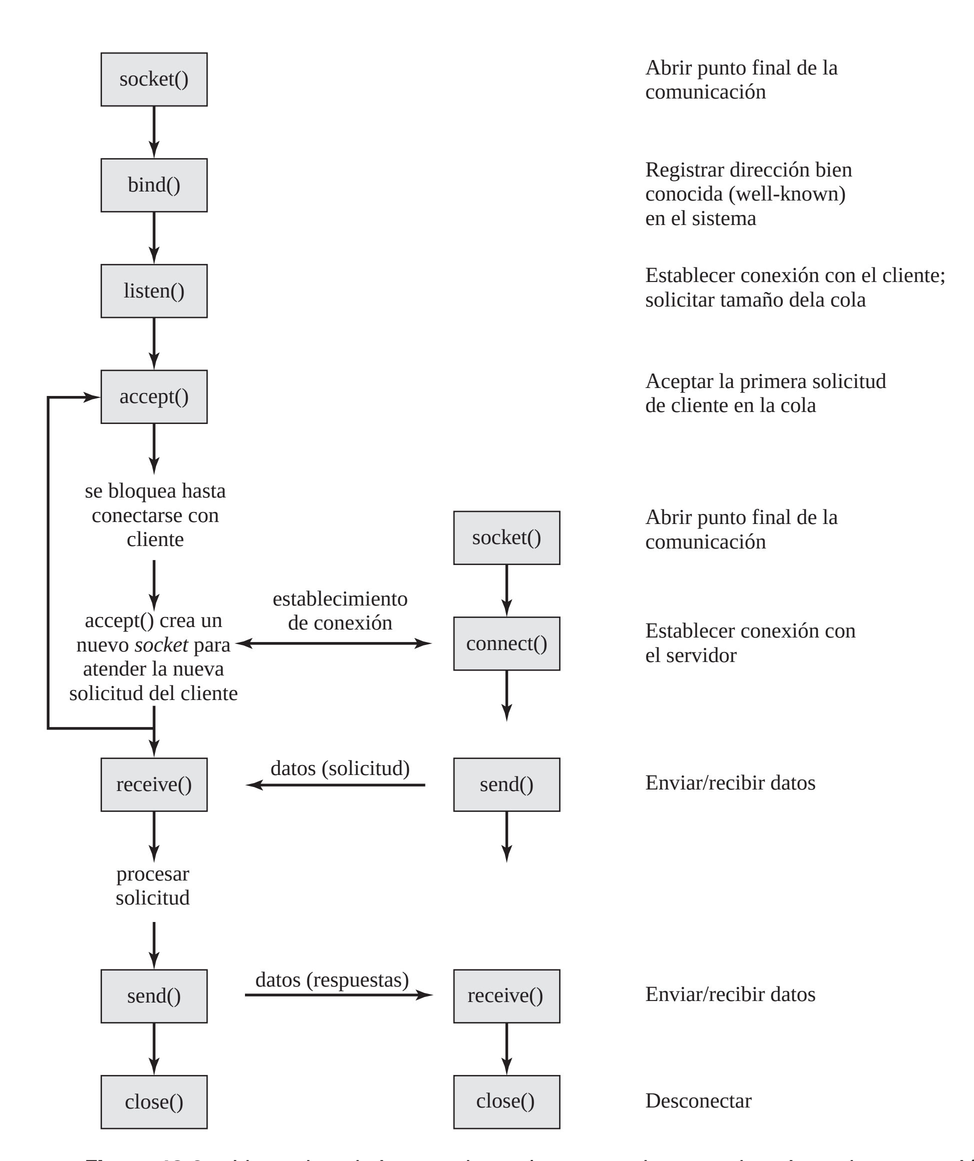


Figura 13.6. Llamadas al sistema de *sockets* para el protocolo orientado a conexión.

rección de destino (dirección IP y puerto). De forma similar, la llamada `recvfrom()` incluye un parámetro de dirección, que se rellena cuando se reciben los datos.

13.4. REDES EN LINUX

Linux soporta diversas arquitecturas de red, en particular TCP/IP a través de los *Berkeley Sockets*. La Figura 13.7 muestra la estructura general del soporte de Linux para TCP/IP. Los procesos de nivel de usuario interactúan con los dispositivos de red a través de llamadas al sistema de la Interfaz de *sockets*. El módulo de *sockets*, a su vez, interactúa con un paquete software en el núcleo que se encarga de las operaciones de la capa de transporte (TCP y UDP) y del protocolo IP. Este paquete software intercambia datos con el manejador del dispositivo para la tarjeta de interfaz de red.

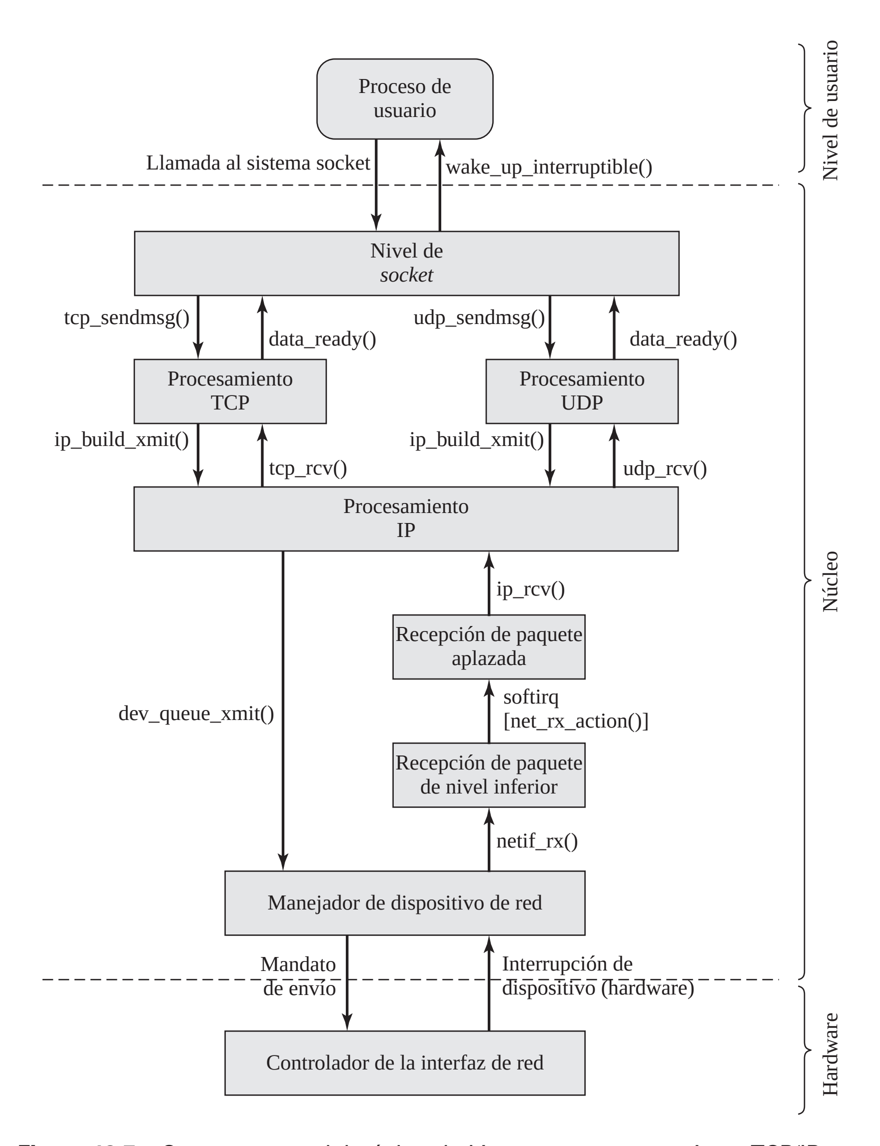


Figura 13.7. Componentes del núcleo de Linux para procesamiento TCP/IP.

Linux implementa los *sockets* como ficheros especiales. Recuerde de la Sección 12.7 que, en sistemas UNIX, un fichero especial no contiene datos pero proporciona un mecanismo para asociar dispositivos físicos a nombres de fichero. Por cada nuevo *socket*, el núcleo de Linux crea un nuevo nodo-i en el sistema de ficheros especial *sockfs*.

La Figura 13.7 representa las relaciones entre diversos módulos del núcleo involucrados en el envío y recepción de bloques de datos basado en TCP/IP. La parte restante de esta sección analiza los servicios de envío y recepción.

ENVÍO DE DATOS

Un proceso de usuario utiliza las llamadas *sockets* descritas en la Sección 13.3 para crear nuevos *sockets*, establecer conexiones con *sockets* remotos y enviar y recibir datos. Para enviar datos, el proceso de usuario escribe datos al *socket* con la siguiente llamada al sistema de ficheros:

```
write(sockfd, mesg, mesglen)
```

donde `mesglen` es la longitud en bytes del *buffer* `mesg`.

Esta llamada activa al método *write* del objeto fichero asociado con el descriptor de fichero `sockfd`. El descriptor de fichero indica si es un *socket* TCP o UDP. El núcleo asigna las estructuras de datos necesarias e invoca a las funciones de nivel de *socket* apropiadas para pasar los datos al módulo TCP o al módulo UDP. Las funciones correspondientes son `tcp_sendmsg()` y `udp_sendmsg()`, respectivamente. El módulo de nivel de transporte crea la estructura de datos de la cabecera TCP o UDP y llama a `ip_build_xmit()` para invocar al módulo de procesamiento de la capa IP. Este módulo crea un *datagrama* IP para la transmisión y lo sitúa en el *buffer* de transmisión del *socket*. A continuación, el módulo de la capa IP llama a `dev_queue_xmit()`, que encola al *buffer* del *socket* para su transmisión a través del manejador del dispositivo de red. Cuando está disponible, el manejador del dispositivo de red transmitirá los paquetes del *buffer*.

RECEPCIÓN DE DATOS

La recepción de datos es un evento impredecible, por lo que se deben utilizar interrupciones. Cuando llega un datagrama IP, el controlador de interfaz de red manda una interrupción hardware al manejador del dispositivo de red. La interrupción activa una rutina de servicio de interrupción que maneja la interrupción. El manejador crea un *buffer* en el núcleo para el bloque de datos entrante y transfiere los datos desde el controlador de dispositivo hasta el *buffer*. El controlador llama a `netif_rx()` para invocar a una rutina de recepción de paquetes. En esencia, la función `netif_rx()` sitúa al bloque de datos entrante en una cola y a continuación emite una petición de interrupción software (`softirq`) de forma que los datos encolados finalmente se procesarán. La acción que se debe llevar a cabo cuando se procesa `softirq` es la función `net_rx_action()`.

Una vez que se ha metido en la cola una `softirq`, el procesamiento de este paquete se detiene hasta que el núcleo ejecuta la función `softirq`, que es equivalente a decir, hasta que el núcleo responda a esta petición de interrupción software y ejecute la función (en este caso, `net_rx_action()`) asociada con esa interrupción sotfware. Hay tres lugares del núcleo donde el núcleo tiene que verificar si hay alguna `softirq` pendiente: cuando se ha procesado una interrupción hardware, cuando un proceso de nivel de aplicación invoca a una llamada al sistema y cuando se planifica para ejecución un nuevo proceso.

Cuando se realiza la función `net_rx_action()`, recoge el paquete de datos y se lo pasa a un manejador de paquetes IP a través de la llamada **ip_rcv**. El manejador de paquetes IP procesa la cabecera IP y a continuación utiliza **tcp_rcv** o **udp_rcv** para invocar al módulo de procesamiento de la capa de transporte. El módulo de la capa de transporte procesa la cabecera de la capa de transporte y pasa los datos al usuario a través de la interfaz de *Sockets* por medio de una llamada **wake_up_interruptible()**, que despierta al proceso receptor.

13.5. RESUMEN

La funcionalidad de las comunicaciones que necesitan las aplicaciones distribuidas es bastante compleja. Esta funcionalidad generalmente se implementa como un conjunto estructurado de módulos. Los módulos se disponen en capas situadas de forma vertical. Cada capa proporciona una parte de la funcionalidad necesaria y depende de la capa inferior. Esta estructura se conoce como arquitectura de comunicaciones.

Una motivación para el uso de este tipo de estructura es que facilita la tarea de diseño e implementación. Para cualquier paquete grande de software es una práctica estándar dividir la funcionalidad en módulos que se pueden diseñar e implementar de forma separada. Después de diseñar e implementar cada módulo, se puede probar. A continuación, los módulos se pueden unir y probar en conjunto. Esta motivación ha llevado a los vendedores de computadores a desarrollar arquitecturas de protocolos de capas propietarios. Un ejemplo es *Systems Network Architecture* (SNA) de IBM.

También se puede utilizar una arquitecura por capas para construir un conjunto de protocolos de comunicación estándar. En este caso, se mantienen las ventajas del diseño modular. Se pueden desarrollar de forma simultánea estándares para protocolos en cada capa de la arquitectura. Esto divide el trabajo, lo hace más manejable y mejora la velocidad del proceso de desarrollo. La arquitectura de protocolos TCP/IP es la arquitectura estándar utilizada para este propósito. Esta arquitectura contiene cinco capas. Cada capa proporciona un porción de todas las funciones requeridas por las aplicaciones distribuidas. El trabajo de desarrollo continúa, especialmente en la capa de aplicación, donde todavía se están definiendo nuevas aplicaciones distribuidas.

13.6. LECTURAS Y SITIOS WEB RECOMENDADOS

[STAL04] proporciona una descripción detallada del modelo TCP/IP y de todos los estándares en cada capa del modelo. [RODR02] es una referencia muy útil en TCP/IP, que cubre el espectro de los protocolos relacionados con TCP/IP de forma concisa.

[DONA01] presenta una excelente introducción al uso de *sockets*; otra buena introducción es [HALL01]. [MCKU96] y [WRIG95] proporcionan detalles sobre la implementación de los *sockets*.

[BOVE03] cubre muy bien las redes Linux. Otras fuentes interesantes son [INSO02a] y [INSO02b].

BOVE03 Bovet, D., y Cesati, M. *Understanding the Linux Kernel.* Sebastopol, CA: O'Reilly, 2003.

DONA01 Donahoo, M., y Clavert, K. *The Pocket Guide to TCP/IP Sockets.* San Francisco, CA: Morgan Kaufmann, 2001.

HALL01 Hall, B. *Beej's Guide to Network Programming Using Internet Sockets.* 2001. **http://www.ecst.csuchico.edu/~beej/guide/net/html/**

INSO02a Insolvibile, G. «Inside the Linux Packet Filter.» *Linux Journal*, Febrero 2002.

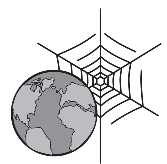
INSO02b Insolvibile,G.«Inside the Linux Packet Filter, Part II.» *Linux Journal*, Marzo 2002.

MCKU96 McKusick, M.; Bostic, K.; Karels, M.; y Quartermain, J. *The Design and Implementation of the 4.4BSD UNIX Operating System.* Reading, MA:Addison-Wesley, 1996.

RODR02 Rodriguez, A., et al. *TCP/IP Tutorial and Technical Overview.* Upper Saddle River: NJ: Prentice Hall, 2002.

STAL04 Stallings, W. *Computer Networking with Internet Protocols and Technology.* Upper Saddle River: NJ: Prentice Hall, 2004.

WRIG95 Wright,G., y Stevens,W. *TCP/IP Illustrated,Volume 2: The Implementation.* Reading, MA: Addison-Wesley, 1995.



SITIOS WEB RECOMENDADOS:

- **Networking Links:** Una excelente colección de enlaces relativos a TCP/IP
- **IPng:** Información sobre IPv6 y temas relacionados

13.7. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

arquitectura de protocolos	Protocolo de Internet (IP)	sockets datagrama
Interfaz de Programación de Aplicaciones (API)	Protocolo de Transferencia de Ficheros (FTP)	sockets *raw*
protocolo	Protocolo Simple de Transferencia de Correo (SMTP)	*sockets stream*
Protocolo de Control de Transmisión (TCP)	puerto	TELNET
Protocolo de Datagramas de Usuario (UDP)	*sockets*	

CUESTIONES DE REPASO

13.1. ¿Cuál es la función principal de la capa de acceso a red?

13.2. ¿Qué tareas se realizan en la capa de transporte?

13.3. ¿Qué es un protocolo?

13.4. ¿Qué es una arquitectura de protocolos?

13.5. ¿Qué es TCP/IP?

13.6. ¿Cuál es el propósito de la interfaz de *sockets*?

PROBLEMAS

13.1. a) Los primeros ministros francés y chino necesitan llegar a un acuerdo por teléfono, pero ninguno habla el idioma del otro. Además, ninguno tiene a mano un traductor que pue-

da traducir al idioma del otro. Sin embargo, los dos primeros ministros tienen traductores a inglés en su personal. Dibuje un diagrama similar al de la Figura 13.8 para describir la situación, y describa la interacción en cada capa.

b) Ahora suponga que el traductor del primer ministro chino puede traducir sólo al japonés, y que el primer ministro francés tiene un traductor a alemán disponible. Se dispone de un traductor entre alemán y japonés en Alemania. Dibuje un nuevo diagrama que refleje esta nueva situación y describa la hipotética conversación de teléfono.

13.2. Enumere las principales desventajas del uso de capas para los protocolos.

13.3. Un segmento TCP que tiene 1500 bits de datos y 160 bits de cabecera se manda a la capa IP, que añade otros 160 bits de cabecera. Este segmento luego se transmite a través de dos redes, cada una de las cuales usa una cabecera de paquete de 24 bits. La red de destino tiene un tamaño máximo de paquete de 800 bits. ¿Cuántos bits, incluyendo cabeceras, se entregan al protocolo de capa de red en el destino?

13.4. ¿Por qué la cabecera TCP tiene un campo de longitud de cabecera y la cabecera UDP no?

13.5. La versión previa de la especificación TFTP, RFC 783, incluía la siguiente sentencia:

Todos los paquetes menos los utilizados para finalización son confirmados individualmente a menos que suceda un *fuera de tiempo* (*timeout*).

La nueva especificación revisada dice:

Todos los paquetes menos los duplicados ACK y los utilizados para finalización son confirmados individualmente a menos que suceda un *fuera de tiempo* (*timeout*).

El cambio se realizó para arreglar un problema conocido como «Aprendiz de Brujo».

Deduzca y explique el problema.

13.6. ¿Cuál es el factor restrictivo en el tiempo requerido para transferir un fichero utilizando TFTP?

13.7. Este capítulo menciona el uso de *Frame Relay* como un protocolo o sistema específico utilizado para conectarse a una red de área extensa. Cada organización tendrá una colección de servicios disponibles (como *Frame Relay*), pero ésto depende del proveedor, coste y equipamiento del cliente. ¿Qué servicios disponibles hay en el área donde reside el lector?

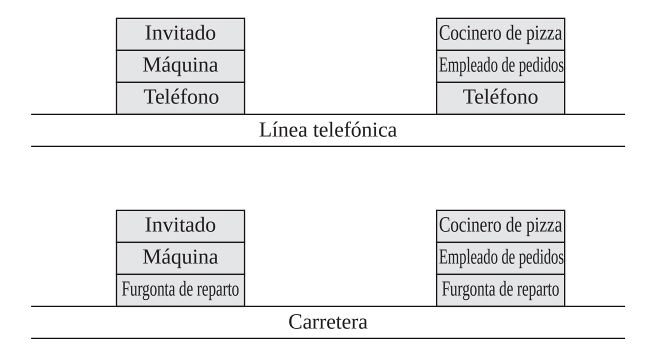


Figura 13.8. Arquitectura para el Problema 13.1.

13.8. Ethereal es un paquete *sniffer* gratuito, que permite capturar el tráfico de una red de área local. Se puede utilizar en una variedad de sistemas operativos y está disponible en **www.ethereal.com.** También se debe instalar el manejador de captura de paquetes WinPcap, que se puede obtener de **http://winpcap.mirror.ethereal.com/**.

Después de comenzar una captura con Ethereal, arranque una aplicación basada en TCP como telnet, FTP o http (navegador Web). ¿Puede ver la siguiente información en su captura?

a) Direcciones origen y destino de la capa 2 (MAC)

b) Direcciones origen y destino de la capa 3 (IP)

c) Direcciones origen y destino de la capa 4 (número de puertos)

13.9. Una aplicación de captura de paquetes o *sniffer*, puede ser una potente herramienta de gestión y seguridad. Utilizando la función de filtrado preconstruida, se puede hacer un seguimiento del tráfico basándose en diferentes criterios y eliminando el resto. Utilice la capacidad de filtrado de Ethereal para:

a) Capturar sólo el tráfico entrante a la dirección MAC de su computador.

b) Capturar sólo el tráfico entrante a la dirección IP de su computador.

c) Capturar sólo las transmisiones basadas en UDP.

APÉNDICE 13A EL PROTOCOLO SIMPLE DE TRANSFERENCIA DE FICHEROS

Este apéndice contiene una visión general del estándar de Internet *Protocolo Simple de Transferencia de Ficheros* (TFTP). El propósito es dar al lector una idea de los elementos de este protocolo.

INTRODUCCIÓN A TFTP

TFTP es mucho más sencillo que el Protocolo de Transferencia de Ficheros (FTP) estándar. No se proporciona control de acceso o identificación de usuario, por lo que TFTP sólo es adecuado para ficheros de acceso público. Por ejemplo, algunos dispositivos sin disco utilizan TFTP para descargar su *firmware* durante el inicio.

TFTP ejecuta sobre UDP. La entidad TFTP que inicia la transferencia lo hace enviando una solicitud de lectura o escritura en un segmento UDP con puerto de destino 69 en el sistema de destino. En el módulo UPD de destino se reconoce este puerto como el identificador del módulo TFTP. Durante la transferencia, cada parte utiliza un identificador de transferencia (TID) como su número de puerto.

PAQUETES TFTP

Las entidades TFTP intercambian mandatos, respuestas y ficheros de datos en forma de paquetes, cada uno de los cuales se transporta en el cuerpo de un segmento UDP. TFTP soporta cinco tipos de paquetes (Figura 13.9); los dos primeros bytes contienen un código de operación (codOp) que identifica el tipo de paquete.

- **RRQ.** El paquete de solicitud de lectura solicita permiso para transferir un fichero desde el otro sistema. Este paquete incluye un nombre de fichero, que es una secuencia de bytes

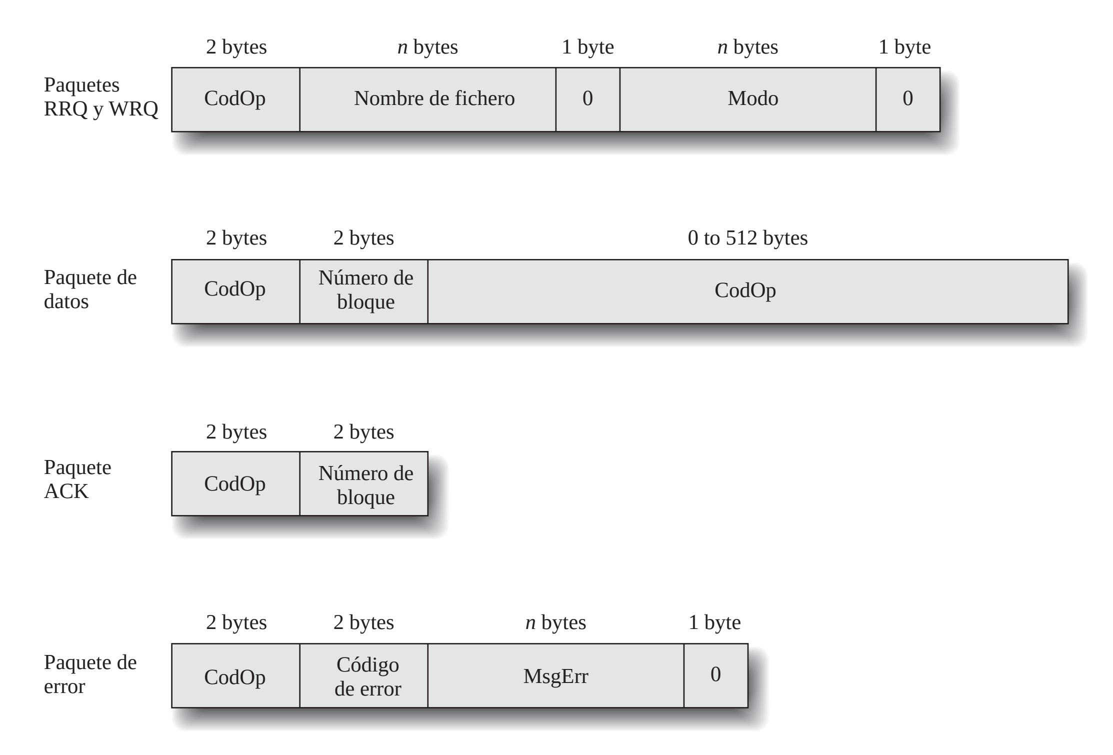


Figura 13.9. Formatos de paquetes TFTP.

ASCII[1] finalizado por un byte cero. El byte cero es la forma como la entidad TFTP receptora sabe cuándo termina el nombre del fichero. El paquete también incluye un campo de modo, que indica si el fichero de datos se interpreta como una cadena de bytes ASCII o como bytes de 8-bits de datos.

- **WRQ.** El paquete de solicitud de escritura solicita permiso para transferir un fichero al otro sistema.
- **Data.** El número de bloque en los paquetes de datos comienza con un uno y se incrementa de uno en uno por cada nuevo bloque de datos. Esta convención permite al programa utilizar un único número para discriminar entre paquetes nuevos y duplicados. El campo de datos tiene una longitud de 0 a 512 bytes. Si tiene 512 bytes de longitud, el bloque no es el último bloque de datos; si tiene longitud de 0 a 511 bytes, indica el final de la transferencia.
- **ACK.** Este paquete se utiliza como confirmación de un paquete de datos o un paquete WRQ. Un ACK de un paquete de datos contiene el número de bloque del paquete de datos que se confirma. Un ACK de un WRQ contiene un cero en el número de bloque.
- **Error.** Un paquete de error puede ser la confirmación de cualquier otro tipo de paquete. El código de error es un número entero que indica la naturaleza del error (Tabla 13.1). El mensaje

[1] ASCII es el *American Standard Code for Information Interchange*, un estándar del *American National Standards Institute*. Designa un patrón único de 7-bits para cada letra, con un octavo bit utilizado para paridad. ASCII es equivalente al *International Reference Alphabet* (*IRA*), definido en *ITU-T Recommendation T.50*. En el sitio web de este libro se encuentra una descripción y una tabla del código IRA.

de error es comprensible y debe estar en ASCII. Como en otras cadenas, se finaliza con un byte cero.

Tabla 13.1. Códigos de error TFTP.

Valor	Significado
0	No definido, ver mensaje de error (si hay)
1	Fichero no encontrado
2	Violación de acceso
3	Disco lleno o ubicación excedida
4	Operación TFTP ilegal
5	ID de transferencia desconocido
6	Fichero ya existe
7	No existe usuario

Todos los paquetes excepto los duplicados ACK (explicado más adelante) o los que se utilizan para finalización, deben ser confirmados. Cualquier paquete se puede confirmar con un paquete de error. Si no hay errores, se usa la siguiente convención. Un WRQ o paquete de datos se confirma con un paquete ACK. Cuando se envía un RRQ, la otra parte responde (en la ausencia de error) comen-

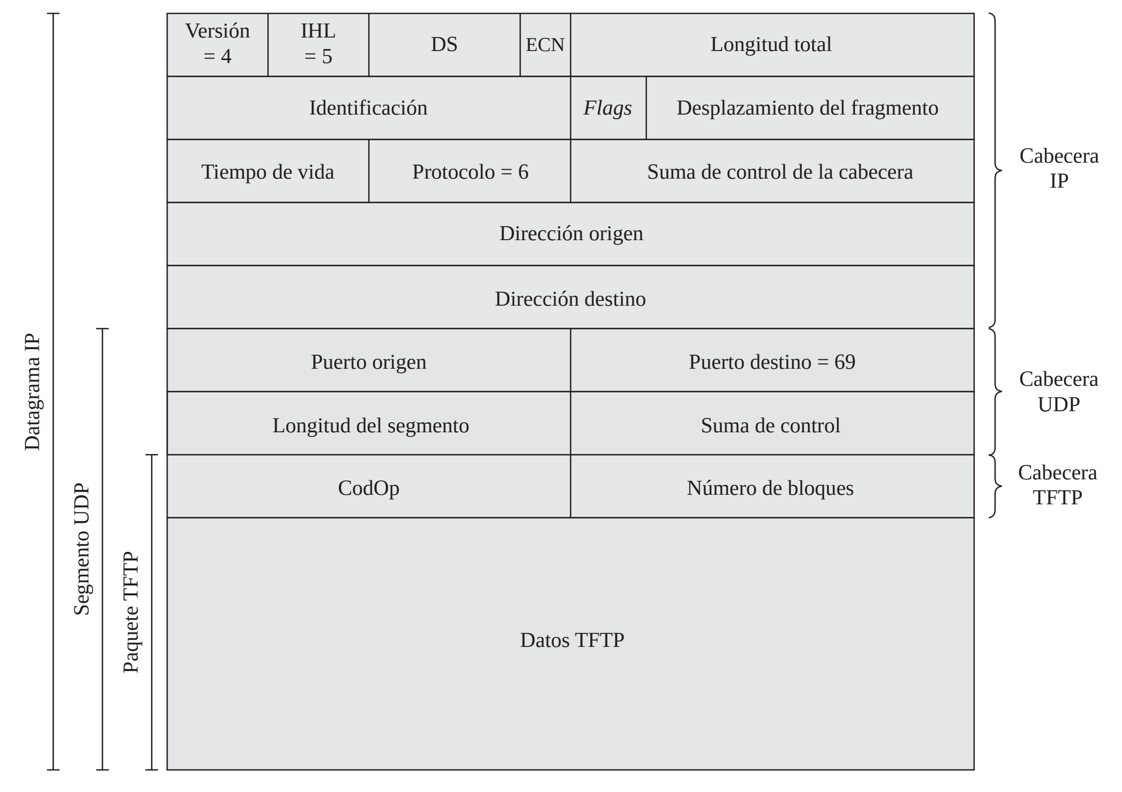


Figura 13.10. Un paquete TFTP en contexto.

zando a enviar el fichero; de esta forma, el primer bloque de datos sirve como confirmación del paquete RRQ. A menos que se complete una transferencia de fichero, cada paquete ACK de una parte se sigue por un paquete de datos de la otra parte, de forma que los paquetes de datos funcionan como confirmaciones. Un paquete de error se puede confirmar con cualquier otro tipo de paquete, dependiendo de las circunstancias.

La Figura 13.10 muestra un paquete de datos TFTP en contexto. Cuando este paquete se le pasa a UDP, UDP le añade una cabecera para formar un segmento UDP. Este segmento se pasa a IP, que le añade una cabecera IP para formar un datagrama IP.

VISIÓN GENERAL DE LA TRANSFERENCIA

El ejemplo mostrado por la Figura 13.11 es de una operación de transferencia de fichero desde A a B. Ni sucede ningún error, ni se detallan las opciones.

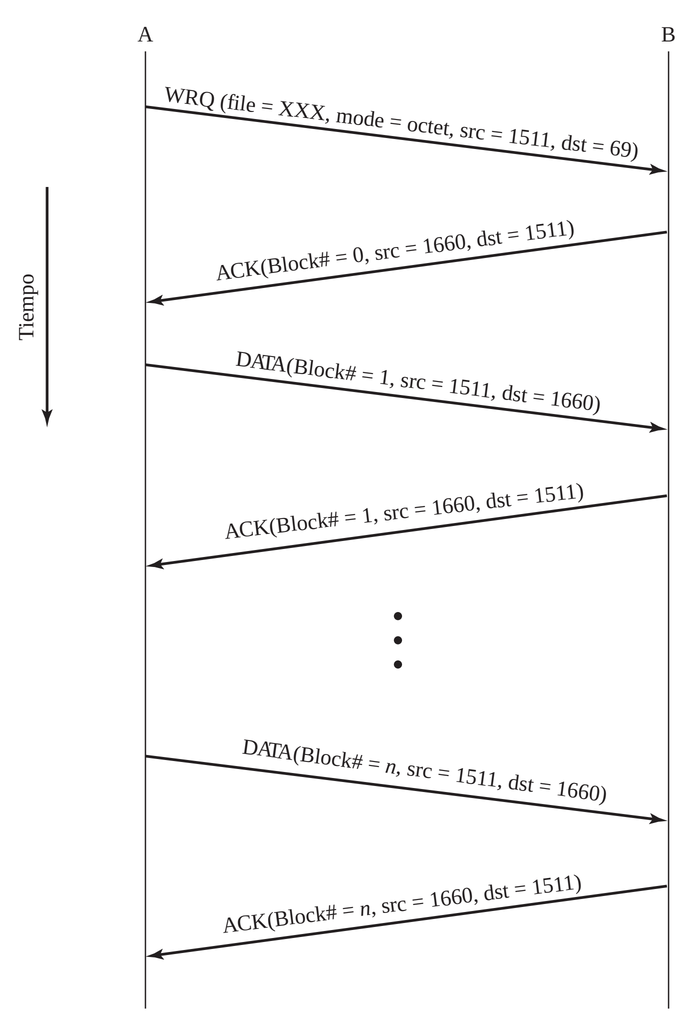


Figura 13.11. Ejemplo de funcionamiento de TFTP.

La operación comienza cuando el módulo TFTP en el sistema A envía una petición de escritura (WRQ) al módulo TFTP del sistema B. El paquete WRQ se transfiere en el cuerpo del segmento UDP. La solicitud de escritura incluye el nombre del fichero (en este caso, XXX), el modo (*octet* o datos *raw*). En la cabecera UDP, el número de puerto de destino es el 69, que le dice a la entidad UDP receptora que este mensaje es de la aplicación TFTP. El número de puerto de origen es un TID seleccionado por A, en este caso 1511. El sistema B está preparado para aceptar el fichero y por eso responde con un ACK con un número de bloque 0. En la cabecera UDP, el puerto de destino es 1511, que permite a la entidad A pasar los paquetes entrantes al módulo TFTP, que puede emparejar este TID con el TID en el WRQ. El puerto de origen es un TID seleccionado por B para esta transferencia de ficheros, en este caso 1660.

Continuando el intercambio inicial, transcurre la transferencia del fichero. La transferencia consiste en uno o más paquetes desde A, cada uno de los cuales se confirma por B. El último paquete de datos contiene menos de 512 bytes, que marca el final de la transferencia.

ERRORES Y RETRASOS

Si TFTP funciona sobre una red o Internet (en oposición a un enlace directo de datos) es posible que se pierdan paquetes. Ya que TFTP funciona sobre UDP, que no proporciona un servicio fiable, necesita haber algún mecanismo en TFTP para tratar los paquetes perdidos. TFTP usa el mecanismo común de *timeout* (fuera de tiempo). Suponga que A envía un paquete a B que requiere confirmación (por ejemplo, un paquete que no sea un ACK duplicado o los utilizados para finalización). Cuando A ha transmitido el paquete, inicia un temporizador. Si el temporizador expira antes de recibir la confirmación desde B, A retransmite el mismo paquete. Si realmente se ha perdido el paquete inicial, la retransmisión será la primera copia de este paquete recibido por B. Si no se perdió el paquete original, pero se perdió la confirmación de B, entonces B recibirá dos copias del mismo paquete y simplemente confirma ambas copias. Debido al uso de los números de bloque, esto no causa ninguna confusión. La única excepción a esta regla es para los paquetes ACK duplicados. El segundo ACK se ignora.

SINTAXIS, SEMÁNTICA Y TEMPORIZACIÓN

En la Sección 5.1, se mencionó que las características principales de un protocolo se pueden clasificar como sintaxis, semántica y temporización. Estas categorías se ven fácilmente en TFTP. Los formatos de varios paquetes TFTP determinan la **sintaxis** del protocolo. La **semántica** del protocolo se muestra en las definiciones de cada tipo de paquete y códigos de error. Para finalizar, la secuencia en que se intercambian los paquetes, el uso de números de bloque y el uso de temporizadores, son todos los aspectos de la **temporización** en TFTP.

CAPÍTULO 14

Procesamiento distribuido, cliente/servidor y *clusters*



Este capítulo comienza examinando algunos de los conceptos básicos en software distribuido, incluyendo la arquitectura cliente/servidor, el paso de mensajes y las llamadas a procedimiento remoto. Luego se examina la cada vez más importante arquitectura cluster.

14.1. COMPUTACIÓN CLIENTE/SERVIDOR

El concepto de la computación cliente/servidor y otros conceptos relacionados, están teniendo cada vez más importancia en los sistemas de tecnología de la información. Esta sección comienza con una descripción general de la arquitectura cliente/servidor. A continuación, se discuten las formas alternativas de organizar las funciones cliente/servidor. Se continúa examinando la consistencia de la *cache* de ficheros y el uso de servidores de fichero. Por último, se introduce el concepto de *middleware*.

¿QUÉ ES LA COMPUTACIÓN CLIENTE/SERVIDOR?

Como toda nueva tendencia en el campo de la computación, la computación cliente/servidor tiene su propia jerga. La Tabla 14.1 enumera algunos de los términos más comúnmente utilizados en la descripción de productos y aplicaciones cliente/servidor.

La Figura 14.1 intenta mostrar la esencia del concepto cliente/servidor. Como el propio término sugiere, un entorno cliente/servidor está formado por clientes y servidores. Las máquinas **cliente** son normalmente simples PCs o estaciones de trabajo que proporcionan una interfaz de fácil manejo al usuario final. Las estaciones cliente suelen utilizar una interfaz gráfica que es más cómoda para los usuarios, e incluyen el uso de ventanas y del ratón. Algunos ejemplos de estas interfaces son los sistemas operativos Microsoft Windows o Macintosh. Las aplicaciones basadas en el cliente están diseñadas para ser fáciles de usar e incluyen aplicaciones tan familiares como una hoja de cálculo.

Tabla 14.1. Terminología cliente/servidor.

Interfaz de programación de aplicaciones (API)
Un conjunto de funciones y programas que permiten a los clientes y servidores intercomunicarse.

Cliente
Un elemento de la red que solicita información, normalmente un PC o estación de trabajo. Puede interrogar a una base de datos o solicitar información de un servidor.

Middleware
Un conjunto de controladores, API, y software adicional que mejoran la conectividad entre una aplicación cliente y un servidor.

Base de datos relacional
Una base de datos en la que el acceso a la información está restringido por la selección de filas que satisfacen todos los criterios de búsqueda.

Servidor
Un computador, normalmente una estación de trabajo de gran potencia, un minicomputador, o un *mainframe*, que almacena la información para los clientes de la red.

Lenguaje Estructurado de Consultas (SQL)
Lenguaje desarrollado por IBM y estandarizado por ANSI que permite acceder, crear, actualizar e interrogar bases de datos relacionales.

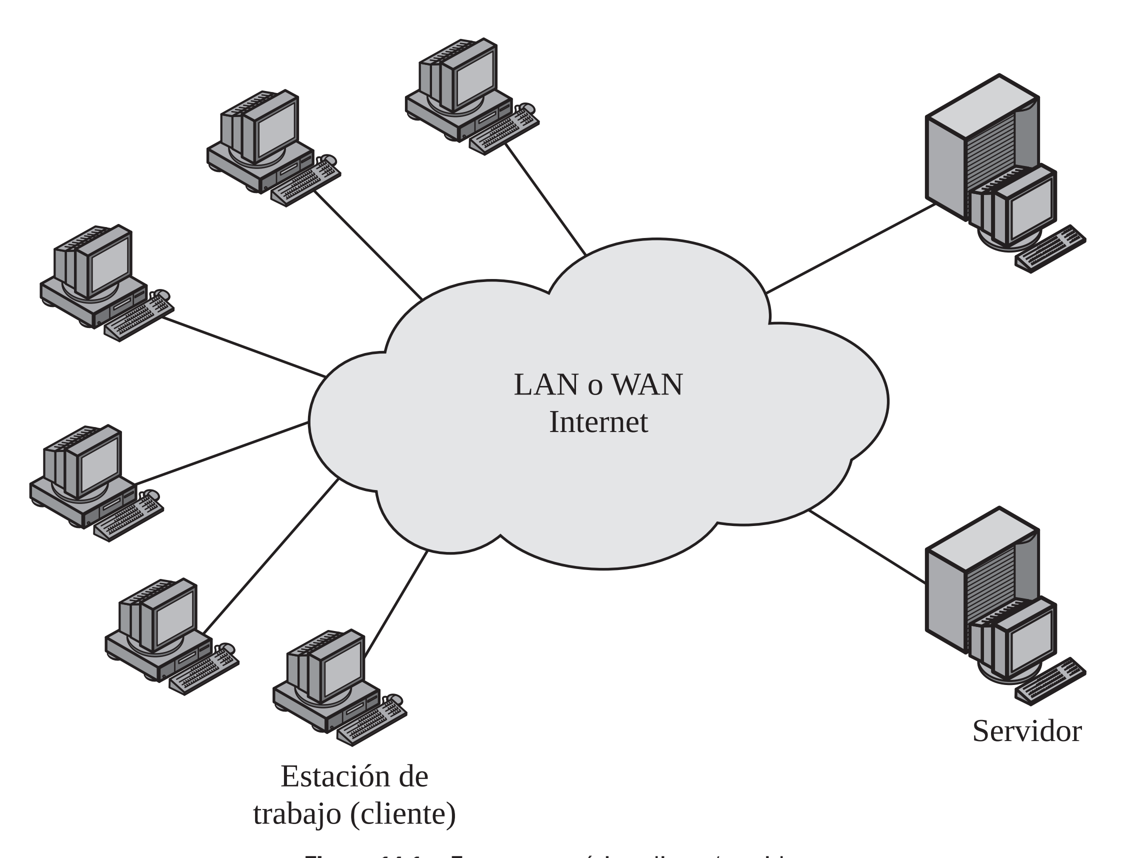


Figura 14.1. Entorno genérico cliente/servidor.

Cada **servidor** en un entorno cliente/servidor proporciona un conjunto de servicios compartidos a los clientes. El tipo más común de servidor es un servidor de base de datos, que normalmente controla una base de datos relacional. El servidor permite a los clientes compartir el acceso a una base de datos y permite el uso de sistemas de computación de alto rendimiento para gestionar la base de datos.

Además de clientes y servidores, el tercer elemento esencial en un entorno cliente/servidor es la **red**. La computación cliente/servidor normalmente es una computación distribuida. Los usuarios, las aplicaciones y los recursos se distribuyen en respuesta a los requisitos de trabajo y se unen a través de una LAN, WAN o Internet.

¿En qué se diferencia la configuración cliente/servidor de otras soluciones de procesamiento distribuido? Existen una serie de características que, juntas, diferencian a la filosofía cliente/servidor de otros tipos de procesamiento distribuido:

- Es imperativo que los usuarios tengan aplicaciones de fácil manejo en sus sistemas. Esto proporciona al usuario un gran control sobre el uso de su computadora y da a los gestores del departamento la capacidad de responder a las necesidades locales.
- A pesar de que las aplicaciones están dispersas, se realiza un esfuerzo por centralizar las bases de datos corporativas, y muchas funciones de utilidad y de gestión de redes. Esto permite a los gestores de la empresa controlar el capital invertido en computación y sistemas de información y permite interoperabilidad de forma que se unen todos los sistemas. Al mismo tiempo

descarga a los departamentos y sucursales de la mayor parte del trabajo de mantener sofisticados sistemas y les permite decidir qué interfaz o máquina utilizar para acceder a los sistemas.

- Existe un compromiso, por parte de las organizaciones de usuarios y vendedores, de mantener un sistema abierto y modular. Esto significa que el usuario tiene más libertad para seleccionar los productos y que puede combinar equipos de diferentes vendedores.
- La red es fundamental. De esta forma, la gestión y la seguridad de la red tienen una prioridad muy alta en la organización y funcionamiento de los sistemas de información.

APLICACIONES CLIENTE/SERVIDOR

La característica fundamental de una arquitectura cliente/servidor es la distribución de las tareas de la aplicación entre el cliente y el servidor. La Figura 14.2 muestra un caso general. Tanto en el cliente como en el servidor, por supuesto, el software básico es el sistema operativo que ejecuta sobre el hardware de la plataforma. La plataforma y el sistema operativo del cliente y del servidor pueden ser diferentes. De hecho, pueden existir diferentes plataformas de clientes y sistemas operativos y diferentes plataformas de servidor en un mismo entorno. Siempre que los clientes y los servidores compartan los mismos protocolos de comunicación y soporten las mismas aplicaciones, estas diferencias de bajo nivel no son relevantes.

Quien permite que interactúen el cliente y el servidor es el software de comunicaciones. El principal ejemplo de este software es TCP/IP. Por supuesto, el objetivo de todo este software de soporte (comunicaciones y sistema operativo) es proporcionar las bases para las aplicaciones distribuidas. De forma ideal, las funciones que realiza una aplicación se pueden dividir entre el cliente y el servidor, de manera que se optimice el uso de los recursos. En algunos casos, dependiendo de las necesidades de la aplicación, la mayor parte del software de aplicación ejecuta en el servidor, mientras que en otros casos, la mayor parte de la lógica de la aplicación está situada en el cliente.

Un factor esencial en el éxito de un entorno cliente/servidor es la forma en que el usuario interactúa con el sistema. De esta forma, es decisivo el diseño de la interfaz de usuario en la máquina cliente. En la mayor parte de los sistemas cliente/servidor, se hace mucho énfasis en proporcionar una **interfaz gráfica de usuario (GUI)** que sea fácil de usar, fácil de aprender y, a la vez, potente y

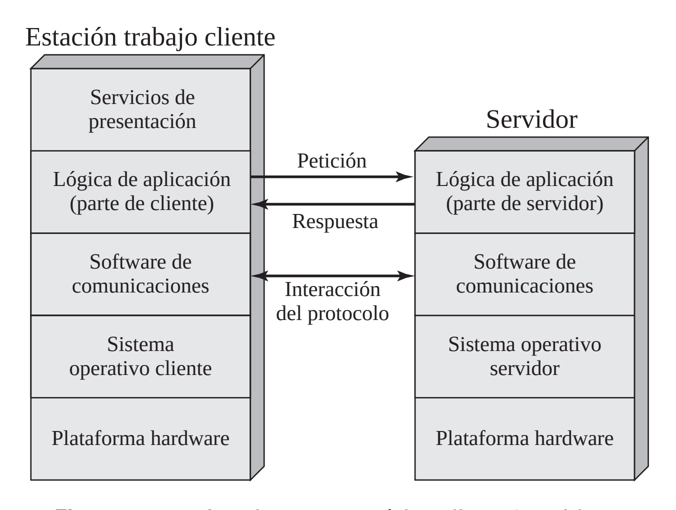


Figura 14.2. Arquitectura genérica cliente/servidor.

flexible. De esta forma, podemos pensar en el módulo de servicios de presentación en la parte del cliente, como el responsable de proporcionar una interfaz de fácil manejo a las aplicaciones distribuidas disponibles en el entorno.

Aplicaciones de base de datos. Para mostrar el concepto de distribuir la lógica de aplicación entre el cliente y el servidor, tomaremos como ejemplo una de las familias más comunes de aplicaciones cliente/servidor: las que hacen uso de bases de datos relacionales. En este entorno, el servidor es básicamente un servidor de base de datos. La interacción entre el cliente y el servidor se realiza a través transacciones, en las que el cliente realiza una petición a la base de datos y recibe una respuesta.

La Figura 14.3 muestra, en términos generales, la arquitectura de este sistema. El servidor es responsable del mantenimiento de la base de datos, para lo que se requiere un complejo sistema gestor de base de datos. En las máquinas clientes se pueden situar diferentes aplicaciones que hacen uso de la base de datos. El «pegamento» que une al cliente y al servidor es el software que permite que el cliente haga peticiones para acceder al servidor de la base de datos. Un ejemplo muy popular es el lenguaje estructurado de consultas (SQL).

La Figura 14.3 sugiere que toda la lógica de aplicación -el software para el análisis de datos- está en la parte cliente, mientras que el servidor sólo se preocupa de la gestión de la base de datos. Que esta configuración sea la apropiada, depende del estilo y del propósito de la aplicación. Por ejemplo, suponga que el propósito general es proporcionar acceso *on-line* para la búsqueda de registros. La Figura 14.4a sugiere cómo podría funcionar. Suponga que el servidor está manteniendo una base de datos con 1 millón de registros (denominados filas en la terminología de bases de datos relacionales), y el cliente quiere realizar una búsqueda que devolverá cero, uno, o como mucho unos pocos registros. El usuario podría buscar estos registros utilizando una serie de criterios de búsqueda (por ejemplo, registros anteriores a 1992; registros de individuos de Ohio; registros de un determinado evento o característica, etc.). Una consulta inicial del cliente puede llevar asociada la respuesta de que hay 100.000 registros que satisfacen estos criterios. El usuario añade información adicional y vuelve a realizar la consulta. Esta vez, la respuesta indica que hay 1000 posibles registros. Finalmente, el cliente manda una tercera consulta con más información. El resultado de la búsqueda lleva a un solo registro, que se le devuelve al cliente.

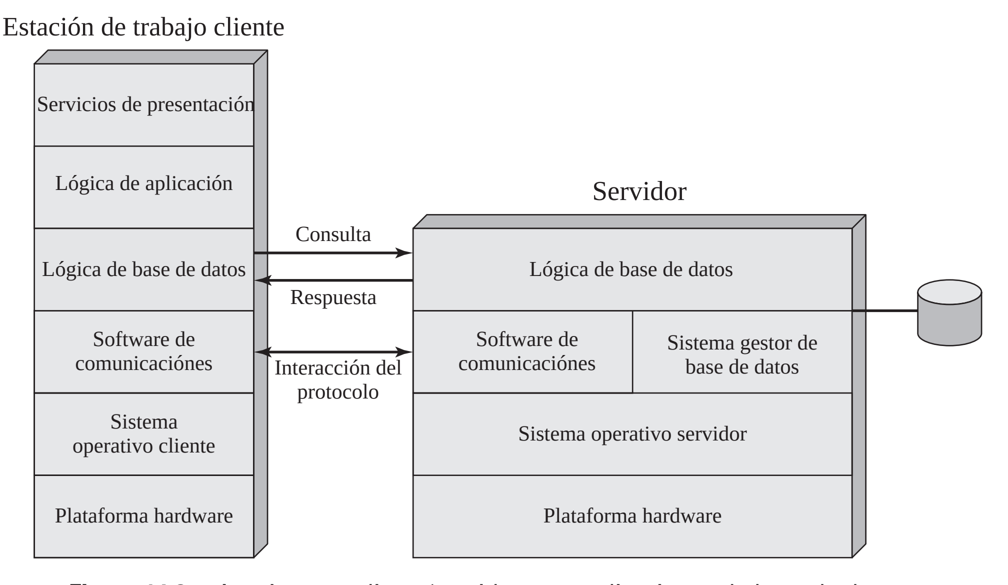


Figura 14.3. Arquitectura cliente/servidor para aplicaciones de base de datos.

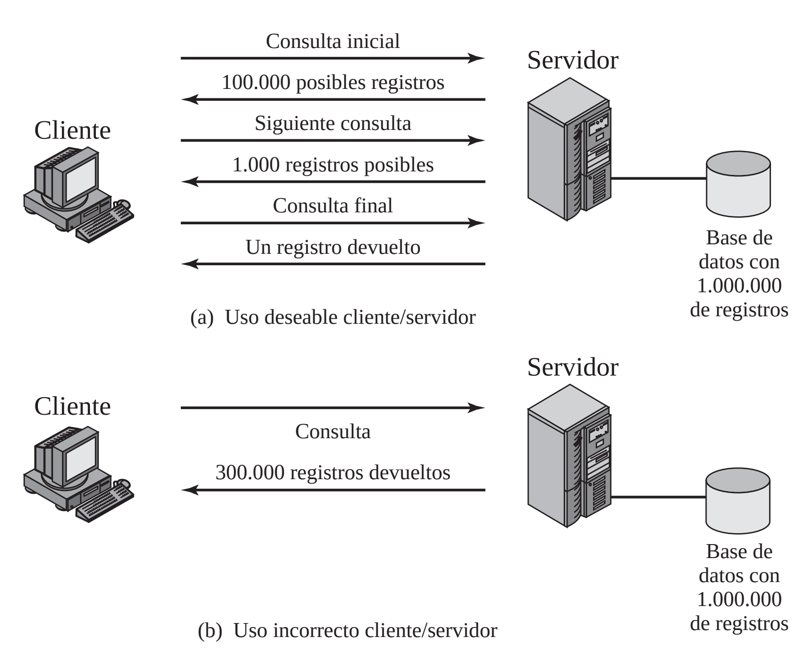


Figura 14.4. Uso de una base de datos en un entorno cliente/servidor.

La aplicación anterior es conveniente para un entorno cliente/servidor por dos razones:

1. Hay un enorme trabajo de búsqueda y ordenamiento en la base de datos. Esto requiere un gran disco o un banco de discos, una CPU con mucha velocidad y una potente arquitectura de E/S. Esta potencia y capacidad no es necesaria, y es demasiado cara, para una estación de trabajo o PC de un usuario.

2. Sería demasiado costoso para la red mover el millón de registros al cliente para que éste realice la búsqueda. Por tanto, no es suficiente que el servidor sea capaz de recuperar registros de la base de datos para el cliente; el servidor necesita tener una lógica que le permita realizar búsquedas para el cliente.

Ahora considere el escenario de la Figura 14.4b, que tiene la misma base de datos con 1 millón de registros. En este caso, la consulta hace que se transmitan 300.000 registros por la red. Esto podría suceder si, por ejemplo, el usuario deseara encontrar el valor total o medio de algunos campos de muchos registros, o incluso de la base de datos completa.

Claramente, este último escenario no es admisible. Una solución a este problema, que mantendría la arquitectura cliente/servidor con todos sus beneficios, es trasladar parte de la lógica de aplicación al servidor. Es decir, se puede equipar al servidor con lógica de aplicación que realice análisis de datos además de recepción y búsqueda de datos.

Clases de aplicaciones cliente/servidor. Dentro del marco general cliente/servidor, hay una serie de implementaciones que dividen el trabajo entre el cliente y el servidor de diferente manera. La Figura 14.5 muestra, en términos generales, algunas de las principales opciones para aplicaciones de bases de datos. Son posibles otras divisiones y, para otra clase de aplicaciones, se podrían realizar caracterizaciones diferentes. En cualquier caso, es bueno observar esta figura y analizar las diferentes posibilidades.

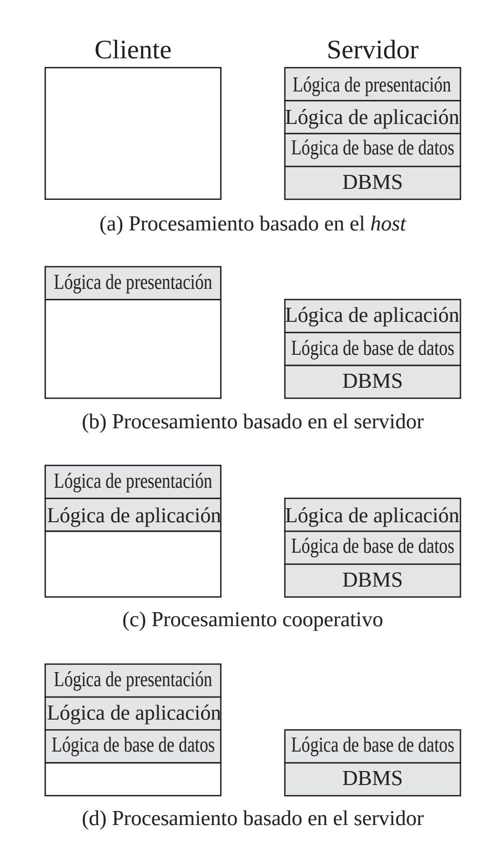


Figura 14.5. Clases de aplicaciones cliente/servidor.

La Figura 14.5 representa cuatro clases:

- **Procesamiento basado en el *host*.** El procesamiento basado en el *host* no es una verdadera computación cliente/servidor tal y como se entiende el término. Se refiere a los entornos *mainframe* tradicionales en los que virtualmente todo el procesamiento se realiza en el *host* central. A menudo, la interfaz de usuario se realiza a través de un interfaz tonto. Incluso si el usuario está empleando una computadora, la estación del usuario se limita al papel de emulador de terminal.

- **Procesamiento basado en el servidor.** La clase más básica de configuración cliente/servidor es en la que el cliente es el principal responsable de proporcionar la interfaz gráfica de usuario, mientras que prácticamente todo el procesamiento se realiza en el servidor. Esta configuración es típica de cuando se comienza a utilizar el entorno cliente/servidor, especialmente en sistemas a nivel de departamento. El razonamiento detrás de estas configuraciones es que la estación de trabajo del usuario es mejor para proporcionar una interfaz de usuario de fácil manejo, y que las bases de datos y las aplicaciones se pueden mantener más fácilmente en sistemas centrales. Aunque el usuario obtiene la ventaja de una mejor interfaz, este tipo de configuraciones no suele generar grandes ventajas en productividad, ni cambios fundamentales en las funciones de negocio soportadas por el sistema.

- **Procesamiento basado en el cliente.** En el otro extremo, prácticamente todo el procesamiento se puede realizar en el cliente, con la excepción de las rutinas de validación de datos y otras

funciones de la lógica de la base de datos que se pueden realizar mejor en el servidor. Generalmente, alguna de las funciones más sofisticadas de la lógica de la base de datos se realizan en la parte cliente. Esta arquitectura es, quizás, el enfoque cliente/servidor más común actualmente en uso. Permite a los usuarios el uso de aplicaciones adaptadas a las necesidades locales.

- **Procesamiento cooperativo.** En una configuración de procesamiento cooperativo, el procesamiento de la aplicación se realiza de forma óptima, beneficiándose de las máquinas cliente y servidora y de la distribución de los datos. Esta configuración es más compleja de configurar y mantener pero, a largo plazo, puede proporcionar mayor productividad a los usuarios y mayor eficiencia de red que otros enfoques cliente/servidor.

Las Figuras 14.5c y d se corresponden con las configuraciones en que gran parte de la carga está en la parte cliente. Este modelo denominado **cliente pesado** (*fat client*) se ha popularizado gracias a herramientas de desarrollo de aplicaciones como *PowerBuilder* de Sybase Inc. o *SQL Windows* de Gupta Corp. Las aplicaciones desarrolladas con estas herramientas suelen ser departamentales y soportan entre 25 y 150 usuarios [ECKE95]. La principal ventaja de este modelo es que se beneficia de la potencia de los escritorios, descargando a los servidores y haciéndolos más eficientes y menos propensos a ser el cuello de botella.

Existen, sin embargo, varias desventajas en la estrategia de los clientes pesados. Añadir nuevas funcionalidades suele sobrecargar la capacidad de los ordenadores de escritorio, forzando a las compañías a actualizarse. Si el modelo sale del departamento y se incorporan muchos usuarios, la compañía debe instalar redes locales (LAN) de alta capacidad para dar soporte al gran número de transmisiones entre los servidores ligeros y los clientes pesados. Por último, es difícil mantener, actualizar o reemplazar aplicaciones distribuidas entre decenas o centenas de ordenadores.

La Figura 14.5b es representativa de un enfoque de **cliente ligero**. Este enfoque imita al enfoque tradicional centralizado del *host* y es, a menudo, la ruta de migración para pasar las aplicaciones corporativas de los *mainframe* a un entorno distribuido.

Arquitectura cliente/servidor de tres capas. La arquitectura tradicional cliente/servidor implica normalmente dos niveles o capas: la capa del cliente y la capa del servidor. También es habitual una arquitectura de tres capas (Figura 14.6). En esta arquitectura, el software de la aplicación se distribuye entre tres tipos de máquinas: una máquina de usuario, un servidor en la capa central, y un servidor en segundo plano (*backend*). La máquina de usuario es la máquina cliente que hemos estado viendo y que, en el modelo de tres capas, normalmente es un cliente ligero. Las máquinas de la capa central son normalmente una pasarela entre los clientes ligeros y varios servidores de bases de datos en segundo plano. Las máquinas de la capa central pueden convertir protocolos y cambiar de un tipo de consulta de base de datos a otra. Además, las máquinas de la capa central pueden mezclar/integrar los resultados de diferentes fuentes de datos. Finalmente, pueden servir como pasarela entre las aplicaciones de escritorio y las aplicaciones de los servidores en segundo plano, mediando entre los dos mundos.

La interacción entre el servidor de la capa central y el servidor en segundo plano, también sigue el modelo cliente/servidor. De esta forma, el sistema de la capa intermedia actúa de cliente y de servidor.

Consistencia de la cache de ficheros. Cuando se utiliza un servidor de ficheros, el rendimiento de la E/S de ficheros se puede degradar en comparación con el acceso local a ficheros, debido a los retrasos generados por la red. Para reducir este problema de rendimiento, los sistemas individuales pueden utilizar cache de ficheros para almacenar los registros de los ficheros a los que se ha accedido recientemente. Debido al principio de localidad, el uso de una *cache* local de ficheros debería reducir el número de accesos necesarios al servidor remoto.

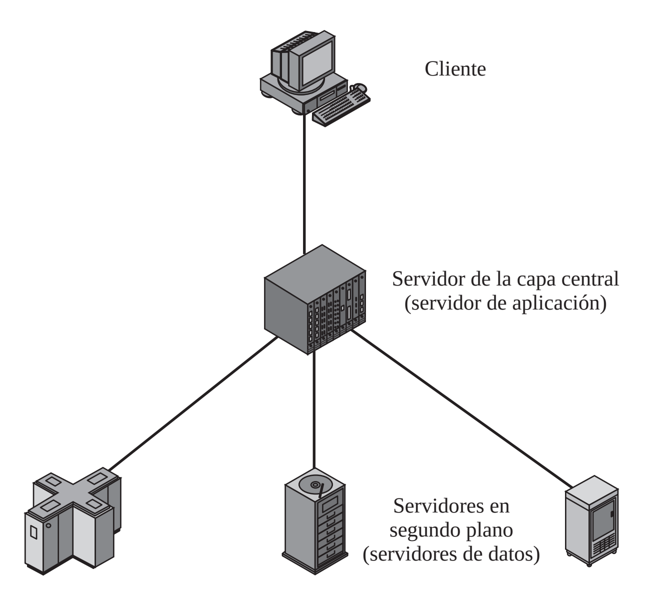


Figura 14.6. Arquitectura cliente/servidor de tres capas.

La Figura 14.7 muestra un mecanismo distribuido típico para la cache de ficheros en una red de estaciones de trabajo. Cuando un proceso realiza un acceso a un fichero, la petición se presenta primero a la cache de la estación de trabajo del proceso («tráfico de fichero»). Si no es satisfactoria, la petición se pasa o al disco local, si el fichero está allí almacenado («tráfico de disco»), o al servidor de ficheros, donde el fichero está almacenado («tráfico de servidor»). En el servidor, se pregunta primero a la cache del servidor, y si no está allí, se accede al disco del servidor. Este enfoque de doble cache se utiliza para reducir el tráfico de comunicación (cache de cliente) y la E/S de disco (cache de servidor).

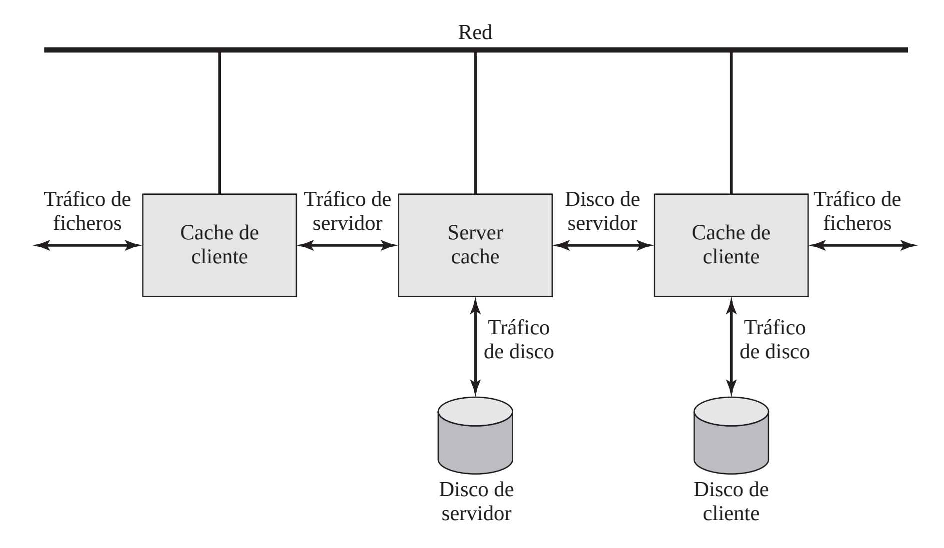


Figura 14.7. Cache de ficheros distribuida en Sprite.

Cuando la cache contiene siempre copias exactas de los datos remotos, decimos que las caches son consistentes. Es posible que las caches se vuelvan inconsistentes cuando se cambian los datos remotos y no se descartan las correspondientes copias locales en la cache. Esto puede suceder si un cliente modifica un fichero que también está en la cache de otro cliente. El problema es doble. Si el cliente adopta la política de escribir inmediatamente en el servidor los cambios de un fichero, cualquier otro cliente que tenga una copia en la cache de la parte del fichero que ha cambiado, tendrá datos obsoletos. El problema es incluso peor si el cliente retrasa la escritura en el servidor. En este caso, el servidor tendrá una versión obsoleta del fichero y las nuevas peticiones de lectura al servidor podrían obtener datos obsoletos. El problema de mantener actualizadas las copias locales de cache es conocido como el problema de la **consistencia de** cache.

El mecanismo más sencillo para la consistencia de cache es utilizar técnicas de bloqueo de ficheros para evitar accesos simultáneos a un fichero por más de un cliente. Esto garantiza la consistencia a costa de rendimiento y flexibilidad. Un mecanismo más potente es el que se proporciona en Sprite [NELS88, OUST88]. Cualquier número de procesos remotos pueden abrir el fichero para lectura y almacenarlo en su propia cache. Pero cuando una solicitud de apertura de fichero pide accesos de escritura y otros procesos tienen el fichero abierto con acceso de lectura, el servidor realiza dos acciones. Primero, notifica al proceso que escribe que, aunque puede mantener una copia local, debe mandar al servidor todos los bloques que se cambien inmediatamente. Puede haber como mucho uno de estos clientes. Segundo, el servidor notifica a todos los procesos lectores que tienen el fichero abierto que el fichero ya no se puede mantener en cache.

MIDDLEWARE

El desarrollo y utilización de los productos cliente/servidor ha sobrepasado con mucho los esfuerzos para estandarizar todos los aspectos de la computación distribuida, desde la capa física hasta la capa de aplicación. Esta falta de estándares hace difícil implementar una configuración cliente/servidor integrada y multivendedor en toda la empresa. Este problema de interoperabilidad debe ser resuelto, ya que gran parte del beneficio del enfoque cliente/servidor está unido a su modularidad y la capacidad de combinar plataformas y aplicaciones para proporcionar una solución de negocio.

Para lograr los verdaderos beneficios del mecanismo cliente/servidor, los desarrolladores deben tener un conjunto de herramientas que proporcionen una manera y estilo de acceso uniforme a los recursos del sistema a través de todas las plataformas. Esto permitirá a los programadores construir aplicaciones que no sólo parezcan iguales en todos los PC y estaciones de trabajo, sino que utilicen el mismo método para acceder a los datos, independientemente de la localización de los mismos.

La forma más común de cumplir estos requisitos es a través del uso de interfaces de programación y protocolos estándares entre la aplicación y el software de comunicaciones y el sistema operativo. Estos interfaces de programación y protocolos estándares se denominan middleware. Con los interfaces de programación estándares resulta sencillo implementar la misma aplicación en varios tipos de servidores y estaciones de trabajo. Esto lógicamente beneficia a los clientes, pero los vendedores también están motivados para proporcionar estos interfaces. La razón es que los clientes compran aplicaciones, no servidores; los clientes sólo elegirán aquellos productos de servidor que ejecuten las aplicaciones que ellos quieren. Los protocolos estandarizados son necesarios para unir a los servidores con los clientes que necesitan acceder a ellos.

Hay diversos paquetes de middleware, que varían desde los muy sencillos a los muy complejos. Lo que tienen todos en común es la capacidad de esconder la complejidad y disparidad de los diferentes protocolos de red y sistemas operativos. Los vendedores de clientes y servidores generalmente proporcionan los paquetes de middleware más populares. De esta forma un usuario puede fijar una estrategia middleware particular y montar equipos de varios vendedores que soporten esa estrategia.

Arquitectura middleware. La Figura 14.8 sugiere el papel del middleware en la arquitectura cliente/servidor. El papel exacto del middleware dependerá del estilo de computación cliente/servidor utilizado. Volviendo a la Figura 14.5, recordamos que hay diferentes enfoques cliente/servidor, dependiendo de la forma en que las funciones de la aplicación se dividen. En cualquier caso, la Figura 14.8 da una buena idea general de la arquitectura involucrada.

Obsérvese que hay componentes tanto cliente como servidor de middleware. El propósito básico del middleware es permitir a una aplicación o usuario en el cliente acceder a una variedad de servicios en el servidor sin preocuparse de las diferencias entre los servidores. Para ver un área específica de aplicación, el lenguaje estructurado de consultas (SQL) se supone que proporciona un acceso estándar a una base de datos relacional por un usuario o aplicación local o remoto. Sin embargo, muchos vendedores de bases de datos relacionales, aunque soportan SQL, han añadido sus propias extensiones. Esto permite a los vendedores diferenciar sus productos, pero también crea potenciales incompatibilidades.

Como ejemplo, considere un sistema distribuido utilizado para dar soporte, entre otras cosas, al departamento de personal. Los datos básicos del empleado, tales como su nombre y dirección, pueden estar almacenados en una base de datos Gupta, mientras que la información de su salario puede estar en una base de datos Oracle. Cuando un usuario en el departamento de personal quiere acceder a un registro en particular, no se quiere preocupar de qué tipo de base de datos contiene los registros. El middleware proporciona una capa de software que permite un acceso uniforme a estos sistemas diferentes.

Es instructivo mirar el papel del middleware desde un punto de vista lógico, más que desde un punto de vista de implementación. Este punto de vista se muestra en la Figura 14.9. El middleware permite la realización de la promesa de la computación cliente/servidor. El sistema distribuido completo se puede ver como un conjunto de aplicaciones y recursos disponibles para los usuarios. Los usuarios no necesitan preocuparse de la localización de los datos o de la localización de las aplicaciones. Todas las aplicaciones operan sobre una interfaz de programación de aplicaciones (API) uniforme. El middleware, que se sitúa entre todas las plataformas cliente y servidor, es el responsable de guiar las peticiones al servidor apropiado.

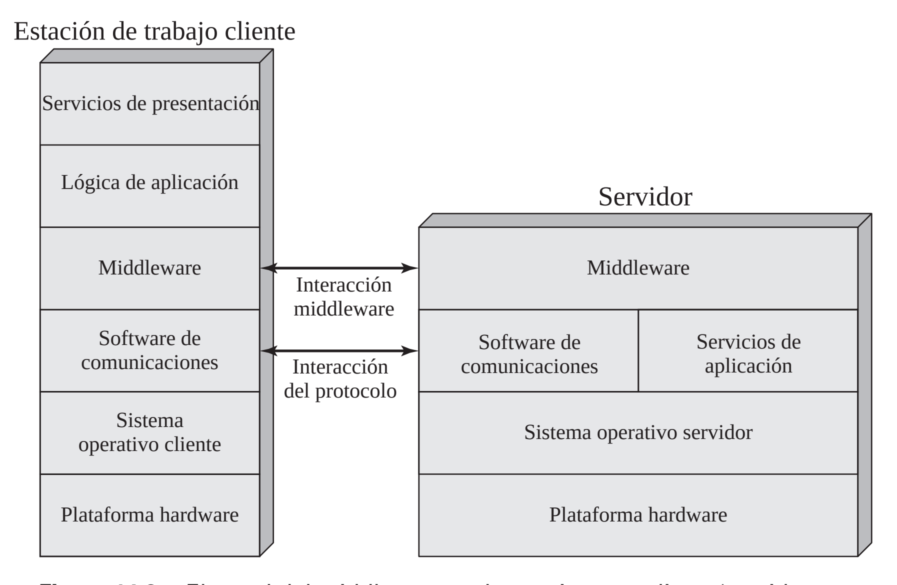


Figura 14.8. El papel del middleware en la arquitectura cliente/servidor.

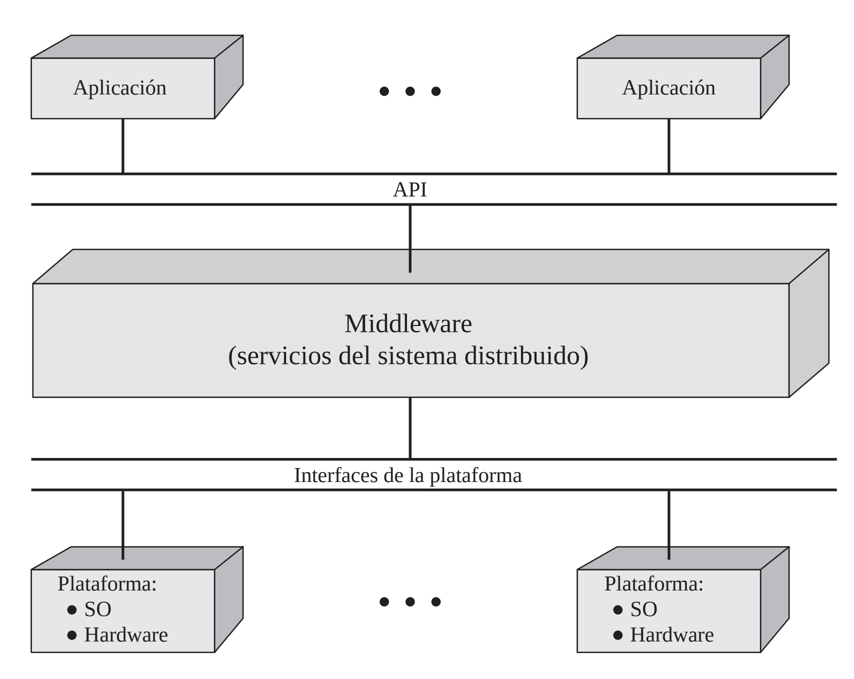


Figura 14.9. Visión lógica del middleware.

Aunque hay una gran variedad de productos middleware, todos se basan típicamente en uno o dos mecanismos: el paso de mensajes y las llamadas a procedimiento remoto. Las dos siguientes secciones examinan estos dos métodos.

14.2. PASO DE MENSAJES DISTRIBUIDO

Normalmente, en los sistemas con procesamiento distribuido, las computadoras no comparten la memoria principal; cada una es un sistema aislado. De esta forma, las técnicas de comunicación entre procesos que se basan en el uso de memoria compartida, tales como los semáforos, no se pueden utilizar. En su lugar, se utilizan técnicas que se basan en el paso de mensajes. En esta sección y en la siguiente, veremos las dos técnicas más comunes. La primera es una aplicación directa de los mensajes tal y como se utilizan en un único sistema. La segunda es una técnica que se basa en el paso de mensajes: las llamadas a procedimiento remoto.

La Figura 14.10a muestra el uso del paso de mensajes para implementar funciones cliente/servidor. Un proceso cliente necesita algún servicio (por ejemplo, leer un fichero, imprimir) y envía un mensaje con la petición de servicio a un proceso servidor. El proceso servidor realiza la petición y envía un mensaje con la respuesta. En su forma más básica, sólo se necesitan dos funciones: Send (mandar) y Receive (recibir). La función Send especifica un destinatario e incluye el contenido del mensaje. La función Receive indica de quién se desea recibir un mensaje (incluyendo la opción «all» —todos—) y proporciona un *buffer* donde se almacenará el mensaje entrante.

La Figura 14.11 sugiere una implementación del paso de mensajes. Los procesos hacen uso de los servicios de un módulo de paso de mensajes. Las peticiones de servicio se pueden expresar en términos de primitivas y parámetros. Una primitiva especifica la función que se desea realizar y los parámetros se utilizan para pasar los datos y la información de control. El formato real de las primitivas

Cliente

Aplicación

Middleware orientado a mensajes (con cola de mensajes)

Transporte

Red

Mensajes específicos de aplicación

Servidor

Aplicación

Middleware orientado a mensajes (con cola de mensajes)

Transporte

Red

(a) Middlewarw orientado a mensajes (con cola de mensajes)

Cliente

Aplicación

Programa esqueleto RPC

Transporte

Red

Invocaciones y respuestas de procedimientos específicos de aplicación

Servidor

Programa esqueleto RPC

Aplicación

Transporte

Red

(b) Llamada a procedimiento remoto

Cliente

Aplicación

Programa esqueleto RPC

Transporte

Red

Solicitud de objetos y respuestas

Mediador de solicitud de objeto

Transporte

Red

Solicitud de objetos y respuestas

Servidor

Servidor de objetos

Transporte

Red

Figura 14.10. Mecanismos middleware.

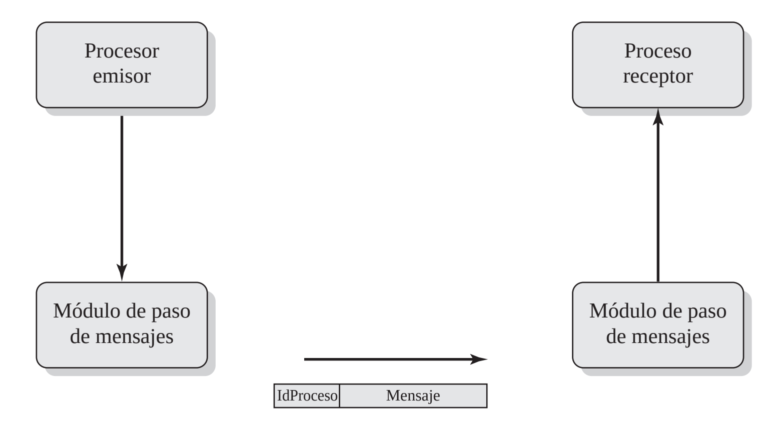


Figura 14.11. Primitivas básicas de paso de mensajes.

depende del software de paso de mensajes que se utilice. Puede ser una llamada a procedimiento o puede ser un mensaje en sí mismo a un proceso que sea parte del sistema operativo.

La primitiva Send se utiliza cuando un proceso quiere enviar un mensaje. Sus parámetros son el identificador del proceso destinatario y el contenido del mensaje. El módulo de paso de mensajes construye una unidad de datos que incluye estos dos elementos. Esta unidad se datos se envía a la máquina que hospeda al proceso destinatario, utilizando algún tipo de servicios de comunicación, como el TCP/IP. Cuando se recibe en el destino la unidad de datos, el servicio de comunicación se la pasa al módulo de paso de mensajes. Este módulo examina el campo ID de proceso y almacena el mensaje en el *buffer* de dicho proceso.

En este escenario, el proceso receptor debe dar a conocer su deseo de recibir un mensaje, designando un área de *buffer* e informando al módulo de paso de mensajes a través de la primitiva Receive. Existe un enfoque alternativo. Cuando el módulo de paso de mensajes recibe un mensaje, señaliza al proceso destinatario con algún tipo de señal Receive y hace disponible el mensaje recibido en un *buffer* compartido.

En la parte restante de esta sección se analizan más aspectos relativos al paso de mensajes.

FIABLE VS. NO FIABLE

Un servicio fiable de paso de mensajes es el que garantiza la entrega si es posible. Estos servicios hacen uso de un protocolo de transporte fiable o de una lógica similar, y realizan comprobación de errores, acuse de recibo, retransmisión y reordenamiento de mensajes desordenados. Ya que se garantiza el envío, no es necesario informar al proceso emisor de que el mensaje ha sido enviado. Sin embargo, podría ser útil informar al proceso emisor vía un acuse de recibo para que sepa que la entrega ya tuvo lugar. En cualquier caso, si falla el envío del mensaje (por ejemplo, fallo persistente de la red, fallo del sistema de destino), se informa del fracaso al proceso emisor.

En el otro extremo, el servicio de paso de mensajes no fiable simplemente envía el mensaje por la red pero no se le indica ni el éxito ni el fracaso. Esta alternativa reduce enormemente la complejidad y la sobrecarga de procesamiento y comunicación del servicio de paso de mensajes. Para las aplicaciones que requieran confirmación de que el mensaje ha sido entregado, la propia aplicación debe enviar y recibir mensajes para satisfacer este requisito.

BLOQUEANTE VS. NO BLOQUEANTE

Con las primitivas no bloqueantes o asíncronas, no se suspende a un proceso como resultado de realizar un Send o Receive. De esta forma, cuando un proceso realiza una primitiva Send, el sistema operativo devuelve el control al proceso tan pronto como el mensaje ha sido puesto en la cola para su transmisión o se ha realizado una copia. Si no se realiza copia, cualquier cambio que el proceso emisor haga al mensaje, antes o durante su transmisión, se realizan bajo la propia responsabilidad del proceso. Cuando el mensaje ha sido transmitido o copiado a un lugar seguro para su transmisión, se interrumpe al proceso emisor para informarle de que el *buffer* puede ser reutilizado. De forma similar, un Receive no bloqueante permite que el proceso continúe ejecutando. Cuando el mensaje llega, se informa al proceso con una interrupción, o bien el propio proceso puede verificar el estado periódicamente.

Las primitivas no bloqueantes proporcionan un uso eficiente y flexible de los servicios de paso de mensajes por parte de los procesos. La desventaja de este mecanismo es la dificultad de chequear y depurar programas que utilicen estas primitivas. Las secuencias irreproducibles y dependientes del tiempo pueden crear problemas delicados y difíciles.

La alternativa es utilizar primitivas bloqueantes o asíncronas. Un Send bloqueante no devuelve el control al proceso emisor hasta que el mensaje ha sido transmitido (servicio no fiable) o hasta que el mensaje ha sido enviado y se ha recibido el acuse de recibo (servicio fiable). Un Receive bloqueante no devuelve el control hasta que el mensaje ha sido situado en su correspondiente *buffer*.

14.3. LLAMADAS A PROCEDIMIENTO REMOTO

Las llamadas a procedimiento remoto son una variante del modelo básico de paso de mensajes. Hoy en día este método es muy común y está ampliamente aceptado para encapsular la comunicación en un sistema distribuido. Lo esencial en esta técnica es permitir a los programas en diferentes máquinas interactuar a través del uso de llamadas a procedimiento, tal y como lo harían dos programas que están en la misma máquina. Es decir, se utilizan las llamadas a procedimiento para acceder a los servicios remotos. La popularidad de este mecanismo se debe a que proporciona las siguientes ventajas:

1. Las llamadas a procedimiento son una abstracción ampliamente aceptada, utilizada y entendida.
2. El uso de llamadas a procedimiento remoto permite especificar las interfaces remotas como un conjunto de operaciones con nombre y tipos de datos dados. De esta forma, la interfaz se puede documentar claramente y los programas distribuidos pueden comprobar estáticamente errores en los tipos de datos.
3. Ya que se especifica una interfaz estandarizada y precisa, el código de comunicación para una aplicación se puede generar automáticamente.
4. Ya que se especifica una interfaz estandarizada y precisa, los desarrolladores pueden escribir módulos cliente y servidor que se pueden mover entre computadoras y sistemas operativos con pocas modificaciones y recodificaciones.

El mecanismo de llamadas a procedimiento remoto se puede ver como un refinamiento del paso de mensajes fiable y bloqueante. La Figura 14.10b muestra la arquitectura general, y la Figura 14.12 proporciona una vista más detallada. El programa llamante realiza una llamada a procedimiento normal con parámetros de su máquina. Por ejemplo,

$$\text{CALL P}(X,Y)$$

donde

P = nombre del procedimiento
X = argumentos pasados
Y = valores devueltos

La intención de invocar a un procedimiento remoto en otra máquina puede ser transparente o no al usuario. En el espacio de direcciones del llamante se debe incluir el esqueleto (*stub*) de un procedimiento P o se debe enlazar dinámicamente en tiempo de llamada. Este procedimiento crea un mensaje que identifica al procedimiento que se está llamando e incluye sus parámetros. De esta forma se envía el mensaje al sistema remoto y se espera una respuesta. Cuando se recibe una respuesta, el procedimiento esqueleto vuelve al programa llamante, proporcionando los valores devueltos.

En la máquina remota, hay otro programa esqueleto asociado con el procedimiento llamado. Cuando llega un mensaje, se examina y se genera un CALL P(X,Y) local. Este procedimiento remoto se llama localmente, de forma que las suposiciones normales de dónde encontrar los parámetros, el estado de la pila y otros, son idénticos al caso de una llamada local.

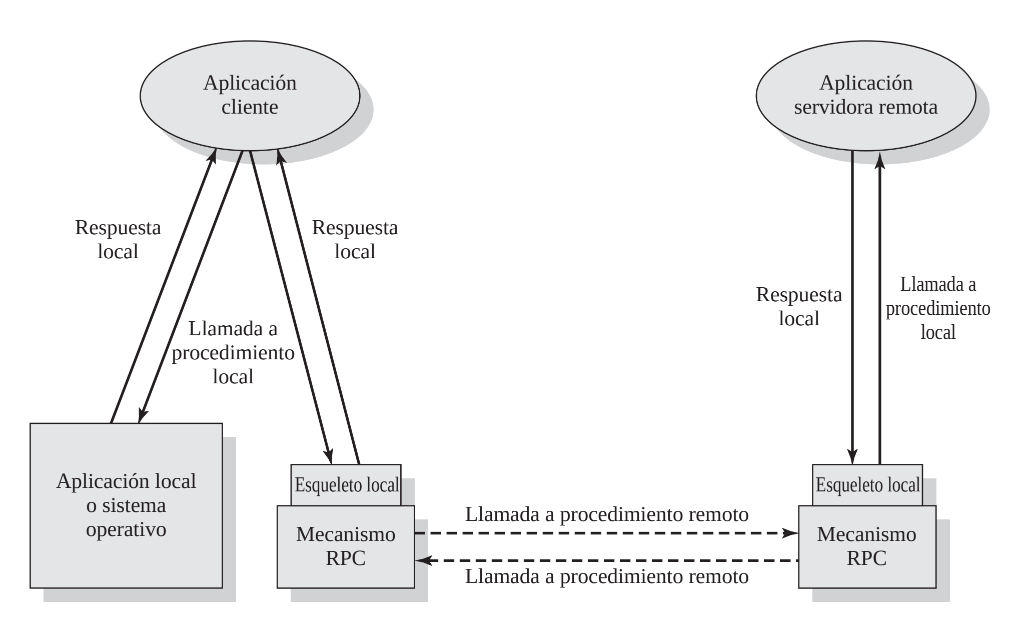


Figura 14.12. Mecanismo de llamadas a procedimiento remoto.

En la parte restante de esta sección se analizan más cuestiones relativas a las llamadas de procedimiento remoto.

PASO DE PARÁMETROS

La mayor parte de los lenguajes de programación permiten que los parámetros se pasen como valores (denominado por valor) o como punteros que contienen los valores (llamado por referencia). Las llamadas por valor son fáciles de implementar en las llamadas a procedimiento remoto: los parámetros se copian en el mensaje y se envían al sistema remoto. Es más difícil implementar las llamadas por referencia. Para cada objeto se necesita un puntero único y válido para todo el sistema. No suele merecer la pena el esfuerzo por la sobrecarga que se genera.

REPRESENTACIÓN DE LOS PARÁMETROS

Otro aspecto es cómo representar los parámetros y los resultados en los mensajes. Si los programas llamado y llamante están escritos en el mismo lenguaje de programación, en el mismo tipo de máquina y con el mismo sistema operativo, entonces el requisito de la representación no es ningún problema. Si hay diferencias en estas áreas, entonces probablemente habrá diferencias en cómo se representan los números o incluso las cadenas de texto. Si se utiliza una arquitectura de comunicación completa, este problema se gestiona en la capa de presentación. Sin embargo, la sobrecarga de este tipo de arquitectura ha llevado al diseño de llamadas a procedimiento remoto con servicios que evitan la mayor parte de la arquitectura de comunicaciones y proporcionan sus propios servicios de comunicación básicos. En este caso, la responsabilidad de la conversión recae sobre el servicio de la llamada a procedimiento remoto (por ejemplo, *véase* [GIBB87]).

El mejor mecanismo para solucionar este problema es proporcionar un formato estándar para los objetos comunes, tales como los enteros, números en coma flotante, caracteres y cadenas de caracte-

res. De esta forma, los parámetros nativos de cada máquina se pueden convertir a y desde la representación estándar.

ENLACE CLIENTE/SERVIDOR

El enlace especifica cómo se establecerá la relación entre un procedimiento remoto y el programa llamante. Se forma un enlace cuando dos aplicaciones han realizado una conexión lógica y están preparados para intercambiar datos y mandatos.

Un **enlace no permanente** significa que la conexión lógica se establece entre los dos procesos en el momento de la llamada a procedimiento remoto y que, tan pronto como se devuelven los valores, se cierra la conexión. Debido a que la conexión requiere que se mantenga información de estado en ambos extremos, consume recursos. El estilo no persistente se utiliza para conservar estos recursos. Por otra parte, la sobrecarga que implica establecer las conexiones, hace que los enlaces no permanentes sean inapropiados para procedimientos remotos que son llamados frecuentemente.

Con los **enlaces persistentes**, una conexión que se establece para una llamada a procedimiento remoto se mantiene después de la finalización de dicha llamada. La conexión puede ser utilizada para futuras llamadas a procedimiento remoto. Si pasa un periodo determinado de tiempo sin actividad en la conexión, se finaliza la conexión. Para aplicaciones que hacen llamadas repetidas a procedimientos remotos, el enlace permanente mantiene la conexión lógica y permite que una secuencia de llamadas utilice la misma conexión.

SÍNCRONOS VS. ASÍNCRONOS

Los conceptos de llamadas a procedimiento remoto síncronas y asíncronas son análogos a los conceptos de mensajes bloqueantes y no bloqueantes. Las llamadas a procedimiento remoto (RPC) tradicionales son síncronas, lo que requiere que el proceso llamante espere hasta que el proceso llamado devuelva el valor. De esta forma, el **RPC síncrono** se comporta como una llamada a subrutina.

El RPC síncrono es fácil de entender y programar porque su comportamiento es predecible. Sin embargo, falla al explotar el paralelismo inherente que tienen los sistemas distribuidos. Esto limita el tipo de interacción que la aplicación distribuida puede tener, dando lugar a rendimientos más bajos.

Para proporcionar mayor flexibilidad, se han implementado algunos servicios **RPC asíncronos** con los que se logra un mayor grado de paralelismo a la vez que se mantiene la familiaridad y simplicidad del RPC [ANAN92]. Los RPC asíncronos no bloquean al llamante; las respuestas se pueden recibir como y cuando sean requeridas, permitiendo de esta forma al cliente ejecutar en paralelo con el servidor.

Un uso típico del RPC asíncrono permite a un cliente invocar a un servidor repetidamente de forma que el cliente tiene varias respuestas en la tubería (*pipeline*) en un determinado momento, cada una con su propio conjunto de datos. La sincronización entre el cliente y el servidor se puede lograr de las siguientes maneras:

1. Una aplicación de una capa superior en el cliente y en el servidor puede iniciar el intercambio y comprobar, al finalizar, que todas las peticiones se han realizado.
2. El cliente puede enviar una cadena de llamadas RPC asíncronas seguida de una llamada RPC síncrona. El servidor responderá a la llamada RPC síncrona sólo después de que se complete todo el trabajo solicitado por las precedentes llamadas RPC asíncronas.

En algunos esquemas, los RPC asíncronos no requieren respuesta del servidor y el servidor no puede enviar un mensaje de respuesta. Otros esquemas requieren o permiten una respuesta, pero el llamante no espera por la misma.

MECANISMOS ORIENTADOS A OBJETOS

A medida que la tecnología orientada a objetos se vuelve más común en el diseño de los sistemas operativos, los diseñadores de la tecnología cliente/servidor han empezado a adoptar este enfoque. En este enfoque, los clientes y los servidores mandan mensajes entre objetos. La comunicación entre objetos se puede basar en una estructura de mensajes o RPC subyacente o puede estar directamente desarrollada sobre los servicios de orientación a objetos del sistema operativo.

Un cliente que necesita un servicio, manda una petición a un mediador de solicitud de objeto (*object request broker*), que actúa como un directorio de todos los servicios remotos disponibles en la red (Figura 14.10c). El mediador llama al objeto apropiado y le transfiere todos los datos relevantes. A continuación el objeto remoto atiende la petición y responde al mediador, que devuelve la información al cliente.

El éxito del enfoque orientado a objetos depende de la estandarización del mecanismo de objetos. Desafortunadamente, hay varios diseños que compiten en esta área. Uno es el Modelo de Componentes de Objeto (*Component Object Model* -COM-) de Microsoft, la base del *Enlazado y Embebido de Objetos* (*Object Linking and Embedding* -OLE-). Un mecanismo competitivo, desarrollado por el Object Management Group, es la Arquitectura Común de Mediador de Solicitud de Objeto (*Common Object Request Broker Arquitecture* -CORBA-), que tiene un gran soporte por parte de la industria. IBM, Apple, Sun y otros muchos vendedores dan soporte a CORBA.

14.4. *CLUSTERS*

Un importante y relativamente reciente diseño de sistema de computación es el *Cluster*. Los *Clusters* son una alternativa al Multiprocesamiento Simétrico (SMP), y son sistemas que proporcionan un alto rendimiento y una alta disponibilidad y que son particularmente atractivos para aplicaciones de servidor. Podemos definir un *cluster* como un grupo de computadoras completas e interconectadas, que trabajan juntas como un recurso de computación unificado y que pueden crear la ilusión de ser una única máquina. El término *computadora completa* significa un sistema que puede ejecutar por sí mismo, aparte del *cluster*; en la literatura, cada computadora de un *Cluster* se denomina *nodo*.

[BREW97] enumera cuatro beneficios que se pueden lograr con los *cluster*. Estos beneficios también se pueden ver como objetivos o requisitos de diseño:

- **Escalabilidad absoluta.** Es posible crear un gran *cluster* que supere la potencia de incluso la mayor de las máquinas. Un *cluster* puede tener decenas o incluso centenas de máquinas, cada una de ellas un multiprocesador.
- **Escalabilidad incremental.** Un *cluster* se configura de tal manera que sea posible añadir nuevos sistemas al *cluster* en pequeños incrementos. De esta forma, un usuario puede comenzar con un sistema pequeño y expandirlo según sus necesidades, sin tener que hacer grandes actualizaciones en las que un pequeño sistema debe ser reemplazado por uno mayor.
- **Alta disponibilidad.** Ya que cada nodo del *cluster* es una computadora en sí mismo, el fallo de uno de los nodos no significa pérdida del servicio. En muchos productos, el software maneja automáticamente la tolerancia a fallos.

- **Relación precio/prestaciones.** A través del uso de bloques de construcción es posible hacer un *cluster* con igual o mayor poder computacional que una única máquina mayor, con mucho menor coste.

CONFIGURACIONES DE LOS *CLUSTERS*

En la literatura, los *Clusters* se clasifican de varias maneras diferentes. Quizás, la configuración más sencilla está basada en si las computadoras del *cluster* comparten acceso a los discos. La Figura 14.13a muestra un *cluster* de dos nodos en que la única interconexión es a través de un enlace de alta velocidad que puede ser utilizado para el intercambio de mensajes y para coordinar la actividad del *cluster*. El enlace puede ser una LAN que está compartida con otras computadoras que no son parte del *cluster*, o puede ser una interconexión dedicada. En el último caso, una o más de las computadoras del *cluster* tendrán un enlace a una LAN o WAN de tal forma que hay una comunicación entre el servidor del *cluster* y los sistemas clientes remotos. Fijarse que en la figura, cada computadora es un multiprocesador. Esto no es necesario, pero mejora tanto el rendimiento como la disponibilidad.

En la clasificación representada en la Figura 14.13, la otra alternativa es un *cluster* con disco compartido. En este caso, sigue habiendo un enlace entre los nodos. Además, hay un subsistema de discos que está directamente enlazado con múltiples computadoras del *cluster*. En la Figura 14.13b,

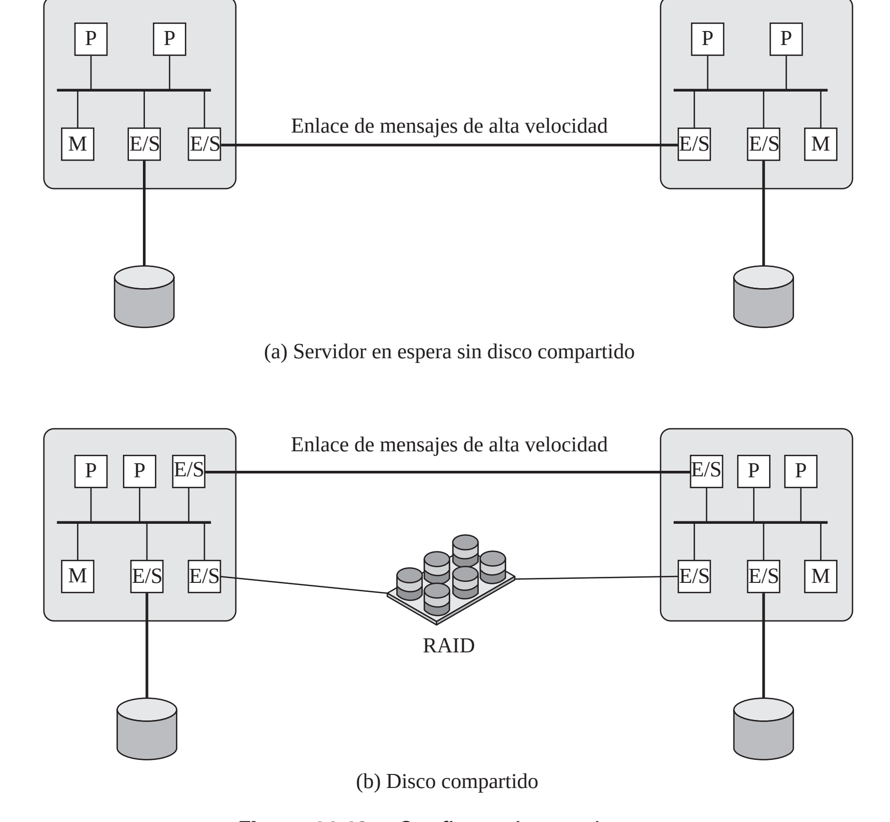


Figura 14.13. Configuraciones *cluster.*

el subsistema de disco común es un sistema RAID. El uso de RAID o de algún sistema de redundancia similar, es común en los *cluster* para que la gran disponibilidad lograda por la presencia de múltiples computadoras, no se comprometa por un disco compartido, lo que podría ser un punto único de fallo.

Mirando a las alternativas funcionales, se puede tener una idea más clara de los tipos de *cluster*. Un documento de Hewlett Packard [HP96] proporciona una clasificación útil junto con sus líneas funcionales (Tabla 14.2), que se discuten a continuación.

Tabla 14.2. Métodos de *cluster*: beneficios y limitaciones.

Método de *Cluster*	Descripción	Beneficios	Limitaciones
Pasivo en Espera	En caso de fallo en el servidor primario, un servidor secundario toma control.	Fácil de implementar.	Alto coste debido a que el servidor secundario no está disponible para procesar otras tareas.
Secundario Activo	El servidor secundario también se utiliza para procesamiento de tareas	Coste reducido porque el servidor secundario puede ser utilizado para procesamiento.	Creciente complejidad.
Diferentes Servidores	Cada servidor tiene sus propios discos. Los datos se copian continuamente del servidor primario al secundario.	Alta disponibilidad.	Alta sobrecarga de red y de servidor debido a las operaciones de copia.
Servidores Conectados a Discos	Los servidores están unidos a los mismos discos, pero cada servidor posee sus propios discos. Si un servidor falla el otro servidor toma control de sus discos.	Sobrecarga de red y de servidores reducida debido a la eliminación de las operaciones de copia.	Normalmente requiere tecnologías de replicación de discos o RAID para compensar el riesgo de fallo de disco.
Servidores Comparten Discos	Varios servidores comparten acceso a disco de forma simultánea.	Baja sobrecarga de red y de servidores. Reducido riesgo de periodos de inactividad causados por fallos de disco.	Requiere software de gestión de cerrojos. Normalmente se utiliza con tecnologías de replicación de discos o RAID.

Un método común, más antiguo, se conoce como pasivo en espera y consiste en tener una computadora realizando todo el proceso, mientras que otra permanece inactiva, esperando tomar control en caso de fallo de la primaria. Para coordinar a las máquinas y su actividad, el sistema manda periódicamente un mensaje de «latido» a la máquina en espera. Si estos mensajes dejan de llegar, la máquina en espera asume que el servidor primario ha fallado y toma el control. Este enfoque mejora la disponibilidad, pero no mejora el rendimiento. Además, si la única información que se intercambian los dos sistemas es el mensaje de control, y si los dos sistemas no tienen discos comunes, el servidor en espera proporciona la funcionalidad necesaria, pero no tiene acceso a las bases de datos manejadas por el servidor primario.

El pasivo en espera no se conoce normalmente como *cluster*. El término *cluster*, en general, se refiere al uso de múltiples computadoras interconectadas, todas ellas realizando procesamiento, a la vez que se mantiene una imagen de sistema único de cara al exterior.

En un enfoque tipo *cluster*, cada computadora es un **servidor diferente** con sus propios discos y no hay discos compartidos entre los sistemas (Figura 14.13a). Esta disposición proporciona alto rendimiento y alta disponibilidad. En este caso, se necesita algún tipo de software de planificación y gestión para asignar las peticiones entrantes de clientes de forma que se balancee la carga y se logre una alta utilización. Es deseable tener la capacidad de recuperación de fallos (*failover*), lo que significa que si una computadora falla durante la ejecución de una aplicación, otra computadora del *cluster* puede tomar el control y finalizar la aplicación. Para que esto suceda, se debe copiar constantemente la información entre los sistemas, de forma que cada sistema tenga acceso a los datos actualizados de otros sistemas. La sobrecarga de este intercambio de datos asegura una alta disponibilidad a costa del rendimiento.

Para reducir la sobrecarga de las comunicaciones, la mayor parte de los *cluster* consisten en servidores conectados a discos comunes (Figura 14.13b). Una variante de este enfoque se denomina **nada compartido** (*shared nothing*). En esta variante, los discos comunes se particionan en volúmenes, y cada volumen pertenece a una computadora. Si falla una computadora, el *cluster* se debe reconfigurar para que otra computadora tome posesión del volumen de la computadora que falló.

También es posible tener múltiples computadoras compartiendo los mismos discos al mismo tiempo (denominada variante **disco compartido** —*shared disk*—), de forma que cada computadora tiene acceso a todos los volúmenes de todos los discos. Esta variante requiere la existencia de algún tipo de servicio de cerrojos que asegure que sólo puede acceder a los datos una sola computadora al mismo tiempo.

ASPECTOS DE DISEÑO DE SISTEMAS OPERATIVOS

Para obtener todas las ventajas de una configuración hardware de un *cluster* se necesitan algunas mejoras en los sistemas operativos.

Gestión de Fallos. La forma de gestionar los fallos depende del método de *cluster* utilizado (Tabla 14.2). En general, para el tratamiento de los fallos se pueden seguir dos enfoques: *clusters* con alta disponibilidad y *clusters* con tolerancia a fallos. Un *cluster* de alta disponibilidad ofrece alta posibilidad de que todos los recursos estén en servicio. Si sucede algún fallo, tal como la caída de un nodo o que se pierda un volumen, se pierden las peticiones en progreso. Cualquier petición perdida que se reintente, será atendida por una computadora diferente del *cluster*. Sin embargo, el sistema operativo del *cluster* no garantiza el estado de las transacciones parcialmente realizadas. Esto se necesita gestionar a nivel de aplicación.

Un *cluster* tolerante a fallos asegura que todos los recursos están siempre disponibles. Esto se logra a través del uso de discos redundantes compartidos y mecanismos para deshacer transacciones incompletas.

La función de intercambiar una aplicación y los datos de un sistema fallido por un sistema alternativo del *cluster* se denomina recuperación de fallos (*failover*). La restauración de aplicaciones y de datos al sistema original una vez que se ha reparado, se denomina restauración de fallos (*failback*). La restauración puede ser automática, pero esto sólo es deseable si el problema está realmente solucionado y es poco probable que vuelva a suceder. De otra forma, la restauración automática puede provocar fallos sucesivos en el sistema inicial, generando problemas de rendimiento y de recuperación.

Equilibrado de carga. Un *cluster* necesita tener la capacidad de equilibrar la carga entre todas las computadoras disponibles. Esto incluye el requisito de que un *cluster* debe ser escalable. Cuando se añade una nueva computadora al *cluster*, el servicio de equilibrado de carga debe incluir automáticamente la nueva computadora en la planificación de las aplicaciones. Los mecanismos del *middleware* deben saber que pueden aparecer nuevos servicios en diferentes miembros del *cluster*, pudiendo migrar de un miembro a otro.

Computación paralela. En algunos casos, el uso eficiente de los *cluster* necesita ejecutar una única aplicación en paralelo. [KAPP00] enumera tres enfoques generales para este problema:

- **Compilación paralela.** Un compilador paralelo determina, en tiempo de compilación, qué partes de la aplicación se pueden ejecutar en paralelo. Estas partes se pueden asignar a computadoras diferentes del *cluster*. El rendimiento depende de la naturaleza del problema y de lo bueno que sea el diseño del compilador.
- **Aplicaciones paralelas.** En este enfoque, el programador escribe la aplicación para que ejecute en un *cluster* y utiliza paso de mensajes para mover datos, según se requiera, entre nodos del *cluster*. Esto supone una gran carga para el programador, pero probablemente es la mejor forma de explotar los *cluster* para algunas aplicaciones.
- **Computación paramétrica.** Este enfoque se puede utilizar si la esencia de la aplicación es un algoritmo que se debe ejecutar un gran número de veces, cada vez con un conjunto diferente de condiciones o parámetros iniciales. Un buen ejemplo es un modelo de simulación, que ejecutará un gran número de diferentes escenarios y luego calculará estadísticas de los resultados. Para que este enfoque sea efectivo, se necesitan herramientas de procesamiento paramétricas para organizar, ejecutar y gestionar los trabajos de forma ordenada.

ARQUITECTURA DE UN CLUSTER

La Figura 14.14 muestra una típica arquitectura *cluster*. Las computadoras están conectadas a una LAN de alta velocidad o conmutador hardware. Cada computadora es capaz de operar independientemente. Además, en cada computadora está instalada una capa de software middleware que permite la operación del *cluster*. El middleware del *cluster* proporciona una imagen única al usuario, conocida como **imagen única del sistema** (*single-system image*). El middleware también podría ser responsable de proporcionar alta disponibilidad, a través del balanceado de carga y de la respuesta a los fallos de los componentes. [HWAN99] enumera los siguientes servicios y funciones deseables en un *cluster*:

- **Un único punto de entrada.** Un usuario se autentifica en el *cluster* y no en una determinada computadora.
- **Una única jerarquía de ficheros.** Los usuarios ven una sola jerarquía de directorios bajo el mismo directorio raíz.
- **Un único punto de control.** Hay un nodo por defecto encargado de gestionar y controlar el *cluster*.
- **Una única red virtual.** Cualquier nodo puede acceder a cualquier otro punto del *cluster*, incluso si la configuración del *cluster* tienen múltiples redes interconectadas. Se opera sobre una única red virtual.
- **Un único espacio de memoria.** La memoria compartida distribuida permite a los programas compartir variables.

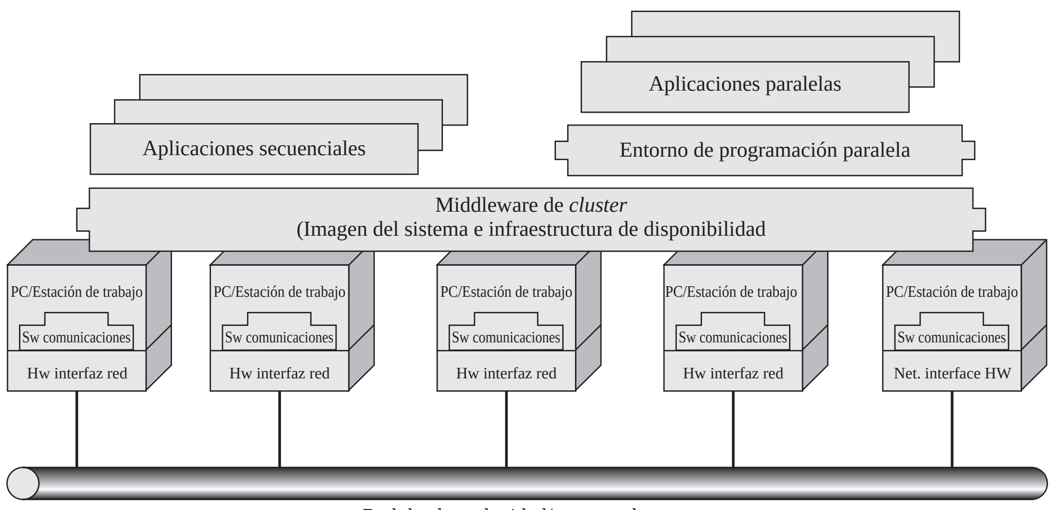


Figura 14.14. Arquitectura de computación *cluster* [BUYY99a].

- **Un único sistema de control de trabajos.** Con un planificador de trabajos en el *cluster*, un usuario puede enviar su trabajo sin especificar la computadora que lo ejecutará.
- **Un único interfaz de usuario.** Todos los usuarios tienen un interfaz gráfico común, independientemente de la estación de trabajo que utilicen.
- **Un único espacio de E/S.** Cualquier nodo puede acceder remotamente a cualquier periférico de E/S o disco, sin conocer su localización física.
- **Un único espacio de procesos.** Se utiliza un esquema uniforme de identificación de procesos. Un proceso en cualquier nodo puede crear o se puede comunicar con cualquier otro proceso en un nodo remoto.
- **Puntos de control.** Esta función salva periódicamente el estado del proceso y los resultados de computación intermedios, para permitir recuperarse después de un fallo.
- **Migración de procesos.** Esta función permite balanceado de carga.

Los últimos cuatro elementos de la lista precedente mejoran la disponibilidad del *cluster*. Los restantes elementos se preocupan de proporcionar una imagen única del sistema.

Volviendo a la Figura 14.14, un *cluster* también incluirá herramientas software para permitir la ejecución eficiente de programas que son capaces de ejecutar en paralelo.

CLUSTER FRENTE A SMP

Tanto los *clusters* como el multiprocesamiento simétrico proporcionan una configuración con múltiples procesadores para dar soporte a aplicaciones con alta demanda. Ambas soluciones están disponibles en el mercado, aunque SMP ha estado presente durante más tiempo.

La principal fuerza del enfoque SMP es que es más fácil de gestionar y configurar que un *cluster*. El SMP está mucho más cercano al modelo original de un solo procesador para los que están escritas prácticamente todas las aplicaciones. El principal cambio requerido para pasar de un uniprocesador a

un multiprocesador es la función de planificación. Otro beneficio del SMP es que normalmente ocupa menos espacio físico y gasta menos energía que un *cluster* comparable. Por último, un beneficio importante es que los productos SMP están bien establecidos y son muy estables.

A largo plazo, sin embargo, las ventajas del *cluster* probablemente le llevarán a dominar el mercado de servidores de alto rendimiento. Los *clusters* son mucho más superiores que SMP en relación a la escalabilidad incremental y absoluta. Los *clusters* son también superiores en términos de disponibilidad, porque todos los componentes del sistema pueden ser altamente redundantes.

14.5. SERVIDOR *CLUSTER* DE WINDOWS

El Servidor *Cluster* de Windows (formalmente conocido por Wolfpack) es un *cluster* de tipo nada compartido, en que cada volumen de disco y otros recursos son propiedad de un único sistema a la vez.

El diseño del Servidor *Cluster* de Windows hace uso de los siguientes conceptos.

- **Servicio *Cluster*.** La colección de software de cada nodo que gestiona toda la actividad específica del *cluster*.
- **Recurso.** Un elemento gestionado por el servicio *cluster*. Todos los recursos son objetos que representan recursos reales en el sistema, incluyendo dispositivos hardware tales como discos o tarjetas de red y elementos lógicos tales como volúmenes lógicos de disco, direcciones TCP/IP, aplicaciones completas y bases de datos.
- **En línea (*online*).** Se dice que un recurso está en línea en un nodo cuando está proporcionando servicio en ese nodo específico.
- **Grupo.** Una colección de recursos gestionada como una unidad. Normalmente, un grupo contiene todos los elementos necesarios para ejecutar una aplicación específica y para que los sistemas cliente se conecten al servicio proporcionado por esta aplicación.

El concepto de grupo es de particular importancia. Un grupo combina recursos en unidades mayores que se pueden manejar más fácilmente, tanto para la recuperación de fallos como para el balanceado de carga. Las operaciones realizadas en un grupo, tales como transferir el grupo a otro nodo, afectan automáticamente a todos los recursos de ese grupo. Los recursos se implementan como bibliotecas dinámicas (DLL) y se gestionan por un monitor de recursos. El monitor de recursos interactúa con el servicio *cluster* a través de llamadas a procedimiento remoto y responde a los comandos del servicio *cluster* para configurar y mover grupos de recursos.

La Figura 14.15 muestra los componentes y sus relaciones en un solo sistema de un *cluster* de Windows. El **gestor de nodo** es responsable de mantener la pertenencia de este nodo al *cluster*. Periódicamente, manda mensajes a los gestores de nodo del resto de los nodos del *cluster*. En caso de que un gestor de nodo detecte la pérdida de mensajes de otro nodo del *cluster*, difunde un mensaje a todo el *cluster*, haciendo que todos los miembros intercambien mensajes para verificar su estado. Si un gestor de nodo no responde, se le quita del *cluster* y sus grupos activos se transfieren a uno o más nodos activos del *cluster*.

El **gestor de la base de datos de configuración** guarda la base de datos de configuración del *cluster*. La base de datos contiene información de recursos, grupos, y pertenencia de grupos a nodos. Los gestores de base de datos de cada uno de los nodos del *cluster* cooperan para mantener una imagen consistente de la información de configuración. Para asegurar que los cambios en la configuración del *cluster* se realizan de forma consistente y correcta, se utiliza software de transacciones tolerante a fallos.

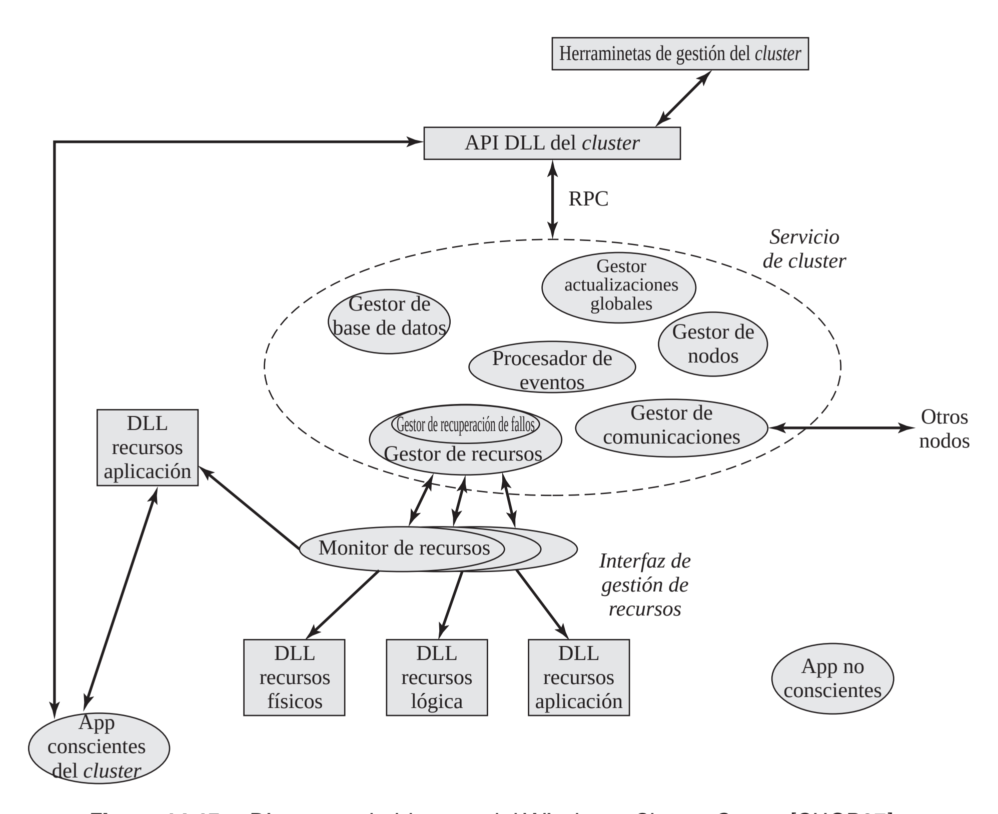


Figura 14.15. Diagrama de bloques del Windows *Cluster Server* [SHOR97].

El **gestor de recursos/gestor de recuperación de fallos** toma todas las decisiones relativas a los grupos de recursos e inicia acciones apropiadas tales como inicializar, reinicializar y recuperar fallos. Cuando se requiere recuperar un fallo, el gestor de recuperación de fallos en el nodo activo coopera para negociar una distribución de los grupos de recursos del sistema que ha fallado en el resto de los sistemas activos. Cuando un sistema se reinicia después de un fallo, el gestor de recuperación de fallos puede decidir hacer regresar algunos grupos a este sistema. En particular, se puede configurar cualquier grupo con un propietario preferente. Si ese propietario falla y luego se reinicia, el grupo se vuelve a llevar al nodo.

El **procesador de eventos** conecta todos los componentes del servicio *cluster*, maneja operaciones comunes y controla la inicialización. El gestor de comunicaciones gestiona el intercambio de mensajes con el resto de los nodos del *cluster*. El gestor global de actualizaciones proporciona un servicio utilizado por otros componentes del *cluster*.

14.6. SUN CLUSTER

Sun Cluster es un sistema operativo distribuido, construido como un conjunto de extensiones del sistema UNIX Solaris. Proporciona una visión unificada; es decir, los usuarios y las aplicaciones ven al *cluster* como una única computadora ejecutando el sistema operativo Solaris.

La Figura 14.16 muestra la estructura global de Sun *Cluster*. Los principales componentes son:

- Soporte de objetos y comunicaciones.

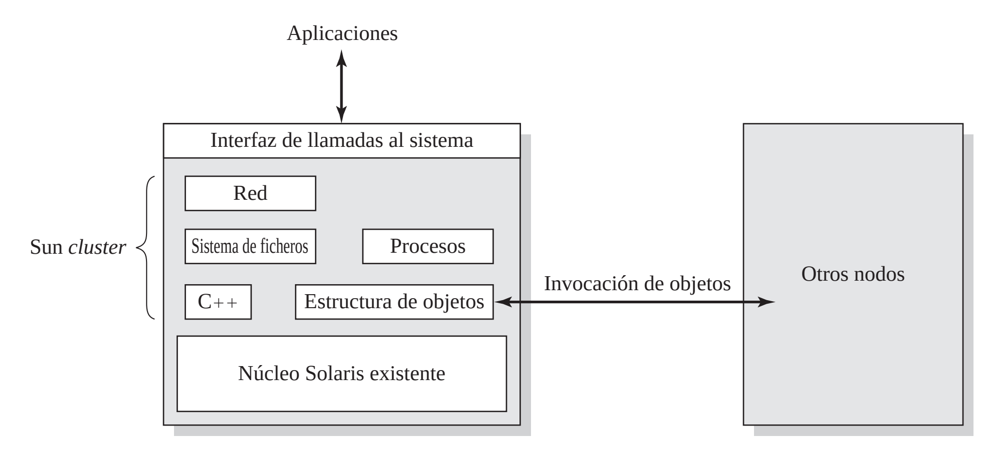


Figura 14.16. Estructura de Sun *Cluster*.

- Gestión de procesos.
- Redes.
- Sistema de ficheros distribuido global.

SOPORTE DE OBJETOS Y COMUNICACIONES

La implementación de Sun Cluster está orientada a objetos. Se utiliza el modelo de objetos de CORBA (*véase* Apéndice B) para definir los objetos y las llamadas a procedimiento remoto (RPC). Se utiliza el Lenguaje de Definición de Interfaces (IDL) de CORBA para especificar los interfaces entre los componentes MC de los diferentes nodos. Los objetos de MC se implementan en el lenguaje orientado a objetos C++. El uso de un modelo de objetos uniforme y de IDL, proporciona un mecanismo para la comunicación entre los procesos, tanto dentro de un nodo como entre ellos. Todo esto está construido encima del núcleo de Solaris sin prácticamente ningún cambio en el núcleo.

GESTIÓN DE PROCESOS

La gestión global de procesos extiende las operaciones de los procesos de forma que la localización de un proceso es transparente al usuario. Sun Cluster mantiene una visión global de los procesos de forma que en el *cluster* hay un identificador único por proceso y que cada nodo puede saber la localización y estado de cada proceso. La migración de procesos está descrita en el Capítulo 15. Un proceso se puede mover de un nodo a otro en su ciclo de vida, con el objetivo de lograr un balanceado de carga y de recuperar fallos. Sin embargo, todos los hilos de un proceso deben estar en el mismo nodo.

REDES

Los diseñadores de Sun Cluster consideraron tres enfoques para el manejo del tráfico de red:

1. Realizar todo el procesamiento del protocolo de red en un único nodo. En particular, para una aplicación basada en TCP/IP, el tráfico entrante (y saliente) podría ir a través de un nodo. Para el tráfico entrante el nodo analizaría las cabeceras TCP/IP y mandaría los datos encapsu-

lados al nodo apropiado. Para el tráfico saliente, el nodo encapsularía los datos de otros nodos con cabeceras TCP/IP. Este enfoque no es escalable a un gran número de nodos, por lo que se descartó.

2. Asignar una dirección única a cada nodo y ejecutar los protocolos de red en cada nodo. Un problema de este enfoque es que la configuración del *cluster* deja de ser transparente al mundo exterior. Otra complicación es la dificultad de la recuperación de fallos cuando una aplicación en ejecución se mueve a otro nodo con una dirección de red diferente.
3. Utilizar un filtro de paquetes para enviar los paquetes al nodo apropiado y realizar el procesamiento del protocolo en dicho nodo. Externamente, el *cluster* parece un solo servidor con una única dirección IP. Las conexiones entrantes (peticiones de clientes), se balancean entre todos los nodos disponibles del *cluster*. Este fue el mecanismo adoptado por Sun Cluster.

El subsistema de red de Sun Cluster tiene tres elementos principales:

1. Los paquetes entrantes se reciben en el nodo que tiene el adaptador de red físicamente instalado; el nodo receptor filtra el paquete y lo envía al nodo objetivo correcto usando la interconexión del *cluster*.
2. Todos los paquetes salientes se envían a través de la interconexión del *cluster* al nodo (o a uno de los múltiples nodos alternativos) que tiene la conexión física de red externa. Todo el procesamiento del protocolo de los paquetes salientes se realiza en el nodo origen.
3. Se mantiene una base de datos de configuración de red para anotar el tráfico de red de cada nodo.

SISTEMA DE FICHEROS GLOBAL

El elemento más importante de Sun Cluster es el sistema de ficheros global, representado en la Figura 14.17, que compara la gestión de ficheros de MC con el esquema básico de Solaris. Ambos se basan en el uso de los conceptos nodo-v y sistema de ficheros virtual.

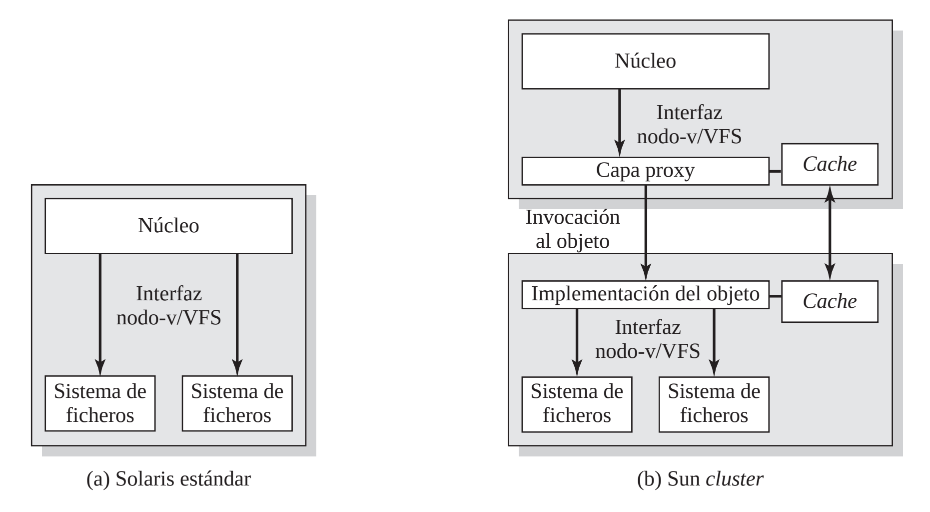


Figura 14.17. Extensiones del sistema de ficheros de Sun *Cluster.*

En Solaris, se utiliza la estructura nodo virtual (nodo-v) para proporcionar una interfaz potente y de propósito general para todos los tipos de sistemas de ficheros. Un nodo-v se utiliza para asociar páginas de memoria con el espacio de direcciones de un proceso y para autorizar acceso a un sistema de ficheros. Mientras que los nodo-i se utilizan para asociar procesos con ficheros UNIX, un nodo-v puede asociar un proceso con un objeto en cualquier tipo de sistema de ficheros. De esta forma, una llamada al sistema no necesita entender el objeto que está manipulando. Sólo necesita saber cómo realizar la llamada orientada a objetos adecuada, utilizando la interfaz del nodo-v. La interfaz del nodo-v acepta mandatos de manipulación de ficheros de propósito general, tales como *read* y *write*, y los traduce en acciones adecuadas al sistema de ficheros en cuestión. Así como los nodo-v se utilizan para describir objetos individuales del sistema de ficheros, las estructuras sistema de ficheros virtual (virtual file system —vfs—) se utilizan para describir el sistema de ficheros completo. La interfaz vfs acepta mandatos de propósito general que funcionan como ficheros completos y los traduce en acciones adecuadas al sistema de ficheros en cuestión.

En Sun Cluster, el sistema de ficheros global proporciona una interfaz uniforme sobre los ficheros distribuidos en el *cluster*. Un proceso puede abrir un fichero localizado en cualquier sitio del *cluster*, y todos los procesos de todos los nodos utilizan la misma ruta para localizar un fichero. Para implementar el acceso global a ficheros, MC incluye un sistema de ficheros *proxy* construido sobre el sistema de ficheros Solaris existente en la interfaz nodo-v. Las operaciones vfs/nodo-v se convierten en invocación a objetos en la capa *proxy* (*véase* Figura 14.17b). El objeto invocado puede residir en cualquier nodo del sistema. El objeto invocado realiza una operación vfs/nodo-v local en el sistema de ficheros subyacente. Ni el núcleo, ni el sistema de ficheros existente, tienen que ser modificados para dar soporte a este entorno global de ficheros.

Para reducir el número de invocaciones a objetos remotos, se utiliza una *cache*. Sun Cluster soporta *cache* de contenidos de ficheros, información de directorios y atributos de ficheros.

14.7. CLUSTERS BEOWULF Y LINUX

En 1994 se inició el proyecto Beowulf con el patrocinio del proyecto de la NASA High Performance Computing and Communications (HPCC) —Computación y Comunicación de Altas Prestaciones—. Su objetivo era investigar el potencial de los *clusters* de PC para realizar tareas importantes de computación, superando las prestaciones de las estaciones de trabajo modernas con un mínimo coste. Hoy en día el enfoque Beowulf está ampliamente implementado y es, quizás, la tecnología *cluster* disponible más importante.

CARACTERÍSTICAS DE BEOWULF

Las principales características de Beowulf incluyen las siguientes [RIDG97]:

- Componentes genéricos disponibles en el mercado.
- Procesadores dedicados (mejor que ciclos disponibles de estaciones de trabajo ociosas).
- Una red privada y dedicada (LAN o WAN o una combinación de redes).
- Ningún componente propio.
- Fácilmente replicable para múltiples vendedores.
- E/S escalable.
- Basado en software gratuito disponible.

- Utiliza herramientas de computación gratuitas con mínimos cambios.
- Retorno del diseño y de las mejoras a la comunidad.

Aunque los elementos software de Beowulf se han implementado en diversas plataformas, la elección más obvia se basa en Linux, y la mayor parte de las implementaciones Beowulf utiliza un *cluster* de estaciones de trabajo Linux sobre PCs. La Figura 14.18 muestra una configuración representativa. El *cluster* tiene una serie de estaciones de trabajo, posiblemente con diferentes plataformas hardware, ejecutando el sistema operativo Linux. El almacenamiento secundario de cada estación de trabajo puede estar disponible para acceso distribuido (para compartición de ficheros distribuida, memoria virtual distribuida y otros usos). Los nodos del *cluster* (los sistemas Linux) se interconectan con una red, normalmente Ethernet. La Ethernet puede estar basada en un solo conmutador (*switch*) o en un conjunto de conmutadores interconectados. Se utilizan productos Ethernet con velocidades estándar (10 Mbps, 100Mbps y 1 Gbps).

BEOWULF SOFTWARE

El software del entorno Beowulf está implementado como una ampliación de las distribuciones Linux gratuitas. La fuente principal de software gratuito de Beowulf está en el sitio **www.beowulf.org,** aunque muchas otras organizaciones también ofrecen herramientas y utilidades Beowulf gratuitas.

Cada nodo de un *Cluster* Beowulf ejecuta su propia copia del núcleo de Linux y puede funcionar como un sistema Linux autónomo. Para dar soporte al *cluster* Beowulf, se realizan extensiones al núcleo de Linux para permitir a los nodos participar en una serie de espacios de nombres globales. Algunos ejemplos del software del sistema Beowulf son los siguientes:

- **Espacio de procesos distribuido de Beowulf (BPROC).** Este paquete permite al espacio de identificadores de proceso expandirse en múltiples nodos de un entorno *cluster* y también proporciona los mecanismos para iniciar procesos en otros nodos. El objetivo de este paquete es proporcionar los elementos clave necesarios para tener una imagen única de sistema del *cluster* Beowulf. BPROC proporciona un mecanismo para iniciar procesos en nodos remotos sin

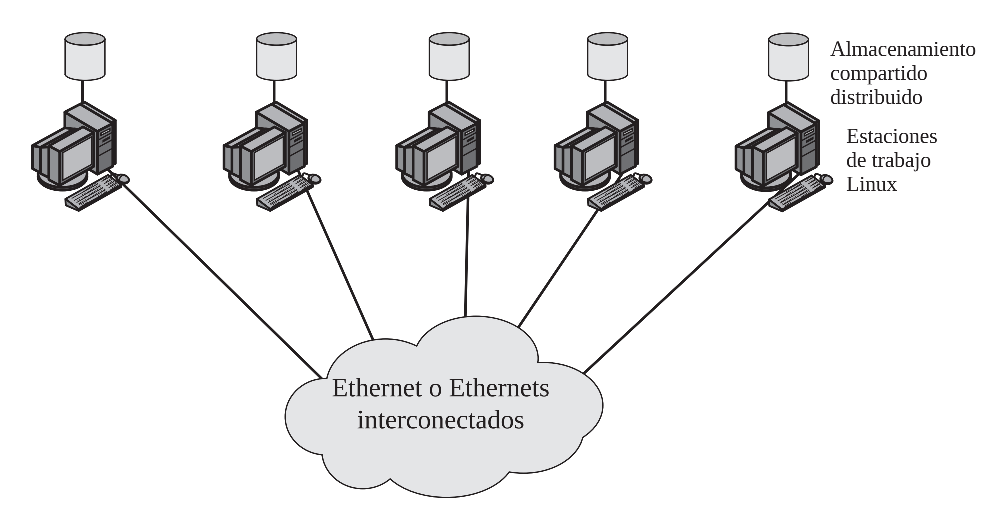


Figura 14.18. Configuración genérica de Beowulf.

haber iniciado sesión en otro nodo y haciendo a todos los procesos remotos visibles en la tabla de procesos del *cluster*.

- **Unión de canales Ethernet Beowulf.** Este es un mecanismo que une múltiples redes de bajo coste en una sola red lógica de mayor ancho de banda. El único trabajo adicional respecto al uso de una sola red es la tarea poco costosa de distribuir los paquetes sobre las colas de los dispositivos disponibles. Este mecanismo permite balancear la carga sobre múltiples Ethernet conectadas a las estaciones de trabajo Linux.
- **Pvmsync.** Este es un entorno de programación que proporciona mecanismos de sincronización y objetos de datos compartidos para los procesos en un *cluster* Beowulf.
- **EnFuzion.** EnFuzion consiste en un conjunto de herramientas para hacer computación paramétrica, tal y como se describió en la Sección 14.4. La computación paramétrica implica la ejecución de un programa como muchos trabajos, cada uno con diferentes parámetros o condiciones iniciales. EnFuzion simula a un conjunto de usuarios robot en un único nodo raíz, cada uno de los cuales accederá a uno de los múltiples clientes que forman el *cluster*. Cada trabajo se configura para ejecutar en un escenario único y programado, con un conjunto adecuado de condiciones iniciales [KAPP00].

14.8. RESUMEN

La computación cliente/servidor es la clave para explotar los sistemas de información y las redes a fin de mejorar significativamente la productividad en las organizaciones. Con el mecanismo cliente/servidor, las aplicaciones de computación se distribuyen a los usuarios en estaciones de trabajo o computadoras personales. Al mismo tiempo, los recursos que pueden y deben ser compartidos se mantienen en sistemas servidores, que están disponibles para todos los clientes. De esta forma, la arquitectura cliente/servidor es una mezcla de computación descentralizada y centralizada.

Normalmente, el sistema cliente proporciona una interfaz gráfica de usuario (GUI) que permite al usuario sacar provecho de múltiples aplicaciones con un entrenamiento mínimo y relativa facilidad. Los servidores mantienen utilidades compartidas, tales como sistemas gestores de base de datos. Las aplicaciones se dividen entre el cliente y el servidor, de forma que se optimice la facilidad de uso y el rendimiento.

El principal mecanismo necesario en un sistema distribuido es la comunicación entre procesos. Se suelen utilizar dos técnicas. El paso de mensajes generaliza el uso de los mensajes en una sola máquina. Se aplican los mismos tipos de convenciones y reglas de sincronización. Otra técnica es el uso de las llamadas a procedimiento remoto. Es una técnica en la que dos programas de diferentes máquinas interactúan usando la sintaxis y la semántica de llamadas a procedimiento. Tanto el programa llamante como el llamado se comportan como si el programa asociado estuviera ejecutando en la misma máquina.

Un *cluster* es un grupo de computadoras completas interconectadas funcionando en conjunto como un recurso de computación unificado que puede crear la ilusión de ser una única máquina. El término *computadora completa* significa un sistema que puede ejecutar por sí mismo, de forma independiente del *cluster*.

14.9. LECTURAS RECOMENDADAS Y SITIOS WEB

[SING99] cubre bastante bien los contenidos de este capítulo. [BERS96] contiene una buena discusión técnica sobre los aspectos de diseño a la hora de situar las aplicaciones en clientes y servidores y

en estrategias middleware; el libro también comenta productos y esfuerzos de estandarización. Una buena visión general de la tecnología y productos middleware es [BRIT04]. [REAG00a] y [REAG00b] tienen un minucioso tratamiento de la computación cliente/servidor y de los enfoques de diseño de redes para dar soporte a dicha computación.

[TANE85] contiene un resumen de los sistemas operativos distribuidos que cubre tanto la comunicación de procesos distribuida como la gestión de procesos distribuida. [CHAN90] proporciona una visión general de los sistemas operativos con paso de mensajes distribuido. [TAY90] es un resumen de los diferentes enfoques que han tomado los sistemas operativos para implementar las llamadas a procedimiento remoto.

[PFIS98] es de lectura obligada para cualquier persona interesada en los *clusters*; el libro cubre los aspectos de diseño del hardware y del software y compara los *clusters* y el SMP. Los *clusters* se tratan en detalle en [BUYY99a] y [BUYY99b]. El primero trata bastante bien Beowulf, que también está muy bien tratado en [RIDG97]. Se pueden encontrar más detalles de Beowulf en [STER99]. El Servidor *Cluster* de Windows se describe en [SHOR97]; [RAJA00] proporciona mayores detalles. Sun Cluster se describe en [SUN99] y [KHAL96].

BERS96 Berson, A. *Client/Server Architecture.* New York: McGraw-Hill, 1996.

BRIT04 Britton,C. *IT Architectures and* Middleware*.* Reading, MA: Addison-Wesley, 2004.

BUYY99a Buyya, R. *High Performance Cluster Computing: Architectures and Systems.* Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High Performance Cluster Computing: Programming and Applications.* Upper Saddle River, NJ: Prentice Hall, 1999.

CHAN90 Chandras, R. «Distributed Message Passing Operating Systems.» *Operating Systems Review*, Enero 1990.

KHAL96 Khalidi, Y., et al. «Solaris MC: A Multicomputer OS.» Proceedings, 1996 USENIX Conference,Enero 1996.

PFIS98 Pfister,G. *In Search of Clusters.* Upper Saddle River, NJ: Prentice Hall, 1998.

RAJA00 Rajagopal, R. *Introduction to Microsoft Windows NT Cluster Server.* Boca Raton, FL: CRC Press, 2000.

REAG00a Reagan, P. *Client/Server Computing.* Upper Saddle River, NJ: Prentice Hall, 2000.

REAG00b Reagan, P. *Client/Server Network: Design, Operation and Management.* Upper Saddle River, NJ: Prentice Hall, 2000.

RIDG97 Ridge, D., et al. «Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs.» *Proceedings, IEEE Aerospace*, 1997.

SHOR97 Short, R.; Gamache, R.; Vert, J. y Massa, M. «Windows NT Clusters for Availability and Scalability.» *Proceedings, COMPCON Spring 97*, Febrero 1997.

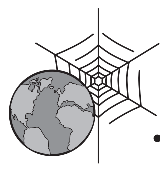
SING99 Singh, H. *Progressing to Distributed Multiprocessing.* Upper Saddle River, NJ: Prentice Hall, 1999.

STER99 Sterling,T., et al. *How to Build a Beowulf.* Cambridge, MA: MIT Press, 1999.

SUN99 Sun Microsystems.«Sun Cluster Architecture:A White Paper.» *Proceedings,IEEE Computer Society International Workshop on Cluster Computing*, Diciembre 1999.

TANE85 Tanenbaum, A. y Renesse, R. «Distributed Operating Systems.» *Computing Surveys*, Diciembre 1985.

TAY90 Tay, B. y Ananda, A. «A Survey of Remote Procedure Calls.» *Operating Systems Review*, Julio 1990.



SITIOS WEB RECOMENDADOS

- **SQL Standards.** Una de las principales fuentes de información sobre el estado del estándar de SQL con documentación actualizada.
- **IEEE Computer Society Task Force on Cluster Computing.** Un foro internacional para promover la investigación y la educación de la computación *cluster*.
- **Beowulf.** Un foro internacional para promover la investigación y la educación de la computación *cluster*.

14.10. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

Beowulf	*cluster*	mensaje
cliente	interfaz de programación de aplicaciones (API)	middleware
cliente ligero		paso de mensajes distribuido
cliente pesado	interfaz gráfico de usuario (GUI)	recuperación de fallos (*failback*)
cliente/servidor	llamadas a procedimiento remoto (RPC)	restauración de fallos (*failover*)
consistencia de *cache* de ficheros		servidor

CUESTIONES DE REPASO

14.1. ¿Qué es la computación cliente/servidor?

14.2. ¿Qué diferencia a la computación cliente/servidor de otros tipos de procesamiento de datos distribuido?

14.3. ¿Cuál es papel de una arquitectura de comunicaciones, tal como TCP/IP, en un entorno cliente/servidor?

14.4. Plantee las razones fundamentales para situar las aplicaciones en el cliente, en el servidor o dividirlas entre el cliente y el servidor.

14.5. ¿Qué son los clientes ligeros y pesados, y cuáles son las diferentes filosofías de los dos enfoques?

14.6. Sugiera pros y contras para las estrategias de clientes ligeros y clientes pesados.

14.7. Explique los fundamentos de la arquitectura cliente/servidor de tres capas.

14.8. ¿Qué es el middleware?

14.9. Ya que tenemos estándares como TCP/IP, ¿por qué se necesita el middleware?

14.10. Enumere algunas ventajas y desventajas de las primitivas bloqueantes y no bloqueantes del paso de mensajes.

14.11. Enumere algunas ventajas y desventajas del enlazado persistente y no persistente en RPC.

14.12. Enumere algunas ventajas y desventajas del RPC síncrono y asíncrono.

14.13. Enumere y defina brevemente cuatro métodos de *cluster* diferente.

PROBLEMAS

14.1. Sea α el porcentaje de código de un programa que se puede ejecutar de forma simultánea por n computadoras de un *cluster*, cada una de ellas utilizando un conjunto diferente de parámetros o condiciones iniciales. Suponga que el código restante debe ser ejecutado secuencialmente por un solo procesador. Cada procesador tiene una tasa de ejecución de x MIPS.

a) Deduzca una expresión para la tasa de MIPS efectiva cuando se usa el sistema únicamente para ejecutar este programa. Hágalo en términos de n, α y x.

b) Si $n = 16$ y $x = 4$ MIPS, determine el valor de α que lleva a un rendimiento del sistema de 40 MIPS.

14.2. Una aplicación está ejecutando en un *cluster* con 9 computadoras. Un programa de análisis comparativo (*benchmark*) tarda un tiempo T en este *cluster*. Además, el 25% del tiempo la aplicación está ejecutando en las 9 computadoras. El tiempo restante, la aplicación ejecuta en una sola computadora.

a) Calcule la ganancia (*speedup*) efectiva bajo las condiciones anteriormente expuestas en comparación con la ejecución del programa en solo una computadora. También calcule α, el porcentaje de código que ha sido paralelizado (programado o compilado) en el programa.

b) Suponga que somos capaces de utilizar 18 computadoras en lugar de 9 para la parte paralelizada del código. Calcule la ganancia efectiva que se logra.

14.3. Se va a ejecutar el siguiente programa FORTRAN en una computadora, y se va a ejecutar una versión paralela en un *cluster* con 32 computadoras.

```
L1:         DO 10 I = 1,1024
L2:            SUM(I)=0
L3:            DO 20 J = 1, I
L4:   20          SUM(I) = SUM(I) + I
L5:   10    CONTINUE
```

Suponga que las líneas 2 y 4 llevan dos ciclos de máquina, incluyendo todas las actividades del procesador y de acceso a memoria. Ignore la sobrecarga generada por las sentencias de control (líneas 1, 3 y 5) y todas las restantes sobrecargas del sistema y conflictos de recursos.

a) ¿Cuál es el tiempo total de ejecución (en tiempo de ciclos de máquina) del programa en una sola computadora?

b) Divida las iteraciones del bucle I entre las 32 computadoras de la siguiente manera: la computadora 1 ejecuta los primeros 32 ciclos (I =1 hasta 32), el procesador 2 ejecuta las siguientes 32 iteraciones y así sucesivamente. ¿Cuál es el tiempo de ejecución y el factor de ganancia en comparación con el apartado (a)? (Note que la carga computacional, mandada por el bucle J, no se reparte entre las computadoras).

c) Explique cómo modificar la paralelización para generar una ejecución paralela balanceada de todo el trabajo de computación sobre los 32 nodos. Por una carga balanceada se entiende un número igual de sumas asignadas a cada computadora respecto a ambos bucles.

d) ¿Cuál es el tiempo mínimo de ejecución sobre las 32 computadoras? ¿Cuál es la ganancia resultante frente a una única computadora?

CAPÍTULO 15

Gestión de procesos distribuidos

◆ ◆

Este capítulo examina unos mecanismos clave utilizados en los sistemas operativos distribuidos. Primero examinamos la migración de procesos, que es el movimiento de un proceso activo de una máquina a otra. Luego, examinamos la cuestión de cómo procesos en diferentes sistemas pueden coordinar sus actividades cuando cada uno está gobernado por un reloj local y cuando hay un retardo en el intercambio de información. Finalmente, exploramos dos aspectos clave en la gestión de procesos distribuidos: la exclusión mutua y el interbloqueo.

◆ ◆

15.1. MIGRACIÓN DE PROCESOS

La migración de procesos es la transferencia de suficiente cantidad del estado de un proceso de un computador a otro para que el proceso ejecute en la máquina destino. El interés en este concepto surge de la investigación en métodos para el equilibrio de carga entre múltiples sistemas interconectados, aunque la aplicación del concepto se extiende ahora más allá de esta área.

En el pasado, sólo unos pocos de los muchos artículos sobre distribución de carga se basaban en implementaciones reales de migración de procesos, lo que incluye la capacidad de expulsar un proceso en una máquina y reactivarlo posteriormente en otra. La experiencia muestra que la migración de procesos expulsiva es posible, aunque con una sobrecarga y complejidad mayores que las anticipadas originalmente [ARTS89a]. Este coste llevó a que algunos observadores concluyesen que la migración de procesos no era práctica. Tales afirmaciones se han probado demasiado pesimistas. Nuevas implementaciones, incluyendo aquellas en productos comerciales, han promovido un interés continuado y nuevos desarrollos en esta área. Esta sección ofrece un repaso.

MOTIVACIÓN

La migración de procesos es deseable en sistemas distribuidos por varias razones [SMIT88,JUL88], incluyendo las siguientes:

- **Compartición de carga.** Moviendo procesos de un sistema muy cargado a otro poco cargado, la carga puede equilibrarse para mejorar el rendimiento global. Datos empíricos sugieren que son posibles mejoras del rendimiento sustanciales [LELA86,CABR86]. Sin embargo, debe tenerse cuidado en el diseño de los algoritmos de equilibrio de carga. [EAGE86] hace notar que cuanta más comunicación sea necesaria para que el sistema distribuido realice el equilibrado, peor será el rendimiento. En [ESKI90] puede encontrarse una exposición sobre este aspecto, con referencias a otros estudios.
- **Rendimiento de las comunicaciones.** Los procesos que interaccionan intensivamente pueden llevarse al mismo nodo para reducir el coste de las comunicaciones mientras dure su interacción. También, cuando un proceso está realizando análisis de datos sobre algún fichero o conjunto de ficheros mayor que el tamaño del proceso, puede ser ventajoso mover el proceso hacia los datos y no viceversa.
- **Disponibilidad.** Se puede necesitar que los procesos de larga duración se muevan para sobrevivir en el caso de fallos que puedan ser conocidos anticipadamente o anticipándose a paradas del sistema planificadas. Si el sistema operativo proporciona la información, un proceso que desea continuar puede bien migrar a otro sistema o asegurarse que podrá rearrancarse en el sistema actual en algún momento posterior.

- **Utilización de facilidades especiales.** Un proceso pueden moverse para conseguir la ventaja de cierta facilidad hardware o software existente únicamente en un nodo particular.

MECANISMOS PARA MIGRACIÓN DE PROCESOS

En el diseño de un servicio para la migración de procesos deben considerarse ciertos aspectos, entre los cuales se encuentran los siguientes:

- ¿Quién inicia la migración?
- ¿Qué parte del proceso se migra?
- ¿Qué sucede con los mensajes y señales pendientes?

Iniciación de la migración. Quién inicie la migración dependerá del objetivo del servicio de migración. Si el objetivo es el equilibrado de carga, entonces cierto módulo del sistema operativo que está monitorizando la carga del sistema será generalmente el responsable de decidir cuándo debe suceder la migración. El módulo será el responsable de expulsar o señalar el proceso que debe emigrar. Para determinar cuándo migrar, el módulo necesitará estar en comunicación con otros módulos en otros sistemas para poder monitorizar el patrón de carga en aquellos sistemas. Si el objetivo es alcanzar ciertos recursos particulares, entonces el proceso puede migrarse a sí mismo cuando surja la necesidad. En este último caso, el proceso debe estar al tanto de la existencia de un sistema distribuido. En el primer caso, la función de migración completa, e incluso la existencia de múltiples sistemas, puede ser transparente para el proceso.

¿Qué se migra? Cuando se migra un proceso, es necesario destruir el proceso en el sistema origen y crearlo en el sistema destino. Esto es el movimiento de un proceso, no su replicación. Así, la imagen del proceso, consistiendo como mínimo el bloque de control de proceso, debe moverse. Además, cualquier vínculo entre este proceso y otros procesos, tales como el paso de mensajes y las señales, deben actualizarse. La Figura 15.1 ilustra estas consideraciones. El Proceso 3 ha migrado de la máquina S para pasar a ser el Proceso 4 en la máquina D. Todos los identificadores de enlace que tengan los procesos (denotados por letras minúsculas) permanecen lo mismo que antes. Es responsabilidad del sistema operativo mover el bloque de control de proceso y actualizar el mapa de los enlaces. La transferencia de un proceso de una máquina a otra es invisible a ambos, el proceso migrado y sus patrones de comunicación.

El movimiento del bloque de control de proceso no presenta complicación. La dificultad, desde el punto de vista de las prestaciones, está en el espacio de direcciones del proceso y en cualquier fichero abierto que se encuentre asignado al proceso. Considérese primero el espacio de direcciones del proceso y asúmase que se utiliza un esquema de memoria virtual (paginación o paginación/segmentación). Pueden considerarse las siguientes estrategias [MILO00]:

- **Ambicioso (todas).** Transferir el espacio de direcciones completo en el momento de la migración. Éste es ciertamente el enfoque más claro. No hay necesidad de dejar atrás en el antiguo sistema ninguna traza del proceso. Sin embargo, si el espacio de direcciones es muy grande y si el proceso no va a necesitar la mayor parte de él, entonces puede ser innecesariamente costoso. Los costes iniciales de la migración pueden ser del orden de minutos. Las implementaciones que proporcionan servicios de punto-de-recuperación (*checkpointing*) y rearranque suelen utilizar este enfoque, porque es más sencillo realizar el punto-de-recuperación y el rearranque si todo el espacio de direcciones está localizado.
- **Precopia.** El proceso continúa la ejecución en el nodo origen mientras el espacio de direcciones se copia en el nodo destino. Las páginas modificadas en el origen durante la operación de

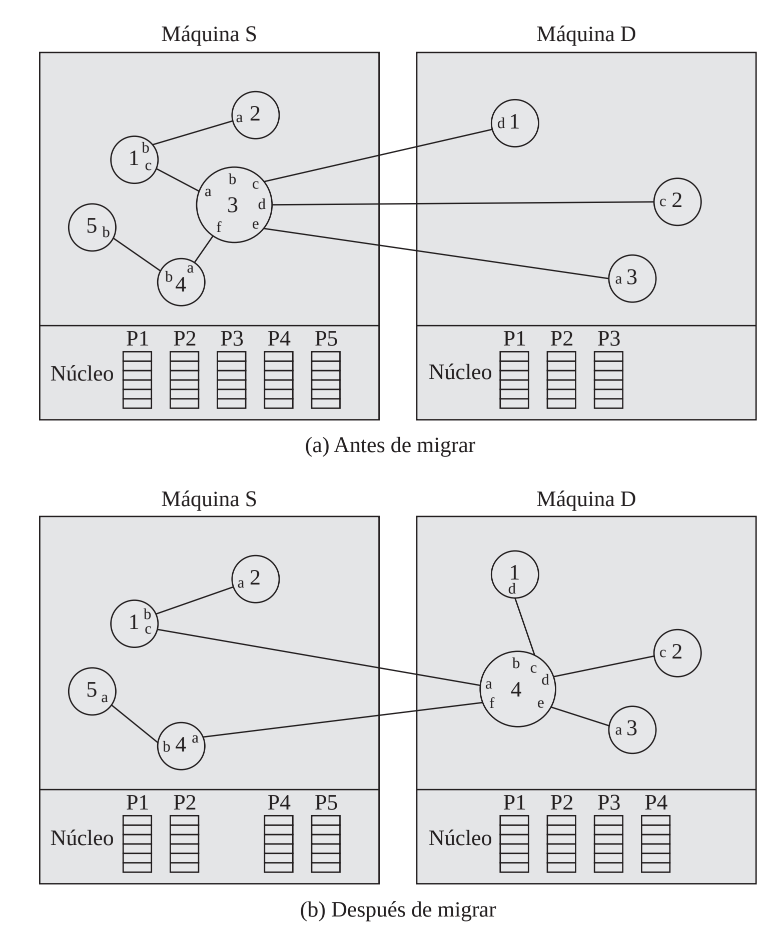


Figura 15.1. Ejemplo de migración de procesos.

copia deben ser copiadas una segunda vez. Esta estrategia reduce el tiempo que el proceso pasa congelado y sin poder ejecutar durante la migración.

- **Ambicioso (sucias).** Transferir sólo aquellas páginas del espacio de direcciones que están en memoria principal y han sido modificadas. Cualquier otro bloque adicional del espacio de direcciones virtuales se transferirá sólo bajo demanda. Esto minimiza la cantidad de datos que se transfieren. Ello requiere, no obstante, que la máquina origen continúe involucrada en la vida del proceso manteniendo tablas de páginas o de segmentos y requiere soporte para la paginación remota.
- **Copiar-al-referenciar.** Esta es una variación del ambicioso (sucias) en la que las páginas son entregadas sólo cuando se les hace referencia. Ésta tiene el menor coste inicial de migración de proceso, yendo de unas pocas decenas a unas pocas centenas de microsegundos.
- **Volcado.** Las páginas del proceso se eliminan de la memoria principal del origen volcando las páginas sucias a disco. Luego se accede a las páginas según se necesiten desde el disco en vez de desde la memoria del nodo origen. Esta estrategia libera al origen de la necesidad de mantener cualquier página del proceso migrado en memoria principal, liberando inmediatamente un bloque de memoria para poder usarse por otros procesos.

Si es posible que el proceso no vaya a usar gran parte de su espacio de direcciones mientras está en la máquina destino (por ejemplo, el proceso va solamente de manera temporal a otra máquina para trabajar sobre un archivo y pronto retornará), entonces tiene sentido utilizar una de las tres últimas estrategias. Por otro lado, si se va a acceder en algún momento a la mayor parte del espacio de direcciones mientras está en la máquina destino, entonces el despiece en bloques de la transferencia del espacio direcciones puede ser menos eficiente que simplemente transferir todo el espacio de direcciones en el momento de la migración, usando una de las dos primeras estrategias.

En muchos casos, puede no ser posible conocer con anticipación cuándo será necesario utilizar o no la mayor parte del espacio de direcciones no residente. No obstante, si el proceso está estructurado como hilos, y si la unidad básica de migración es el hilo en vez del proceso, entonces parece que lo mejor sería una estrategia basada en paginación remota. De hecho, esta estrategia es quizás obligatoria, porque los restantes hilos del proceso se quedan atrás y también necesitan acceder al espacio de direcciones del proceso. La migración de hilos está implementada en el sistema operativo Emerald [JUL89].

Consideraciones similares son de aplicación al movimiento de los ficheros abiertos. Si el fichero está inicialmente en el mismo sistema que el proceso a migrar y si el fichero está bloqueado para acceso exclusivo por tal proceso, entonces puede tener sentido transferir el fichero junto con el proceso. El peligro aquí es que el proceso puede salir solamente de manera temporal y puede no necesitar el fichero hasta que retorne. Por tanto, puede tener sentido transferir el archivo entero solamente después de que el proceso migrado realice una solicitud de acceso. Si un archivo está compartido por múltiples procesos distribuidos, el acceso distribuido al fichero debe mantenerse sin mover el fichero.

Si se permite realizar *cache*, como en el sistema Sprite (Figura 14.7), entonces se introduce una complejidad adicional. Por ejemplo, si un proceso tiene abierto un fichero para escritura y se duplica y se migra el hijo, el fichero estaría entonces abierto para escritura en dos máquinas distintas; el algoritmo de consistencia de *cache* del Sprite dicta que el archivo debe pasar a ser no cacheable en las máquinas en las cuales están ejecutando los dos procesos [DOUG89,DOUG91].

Mensajes y señales. El aspecto final de los enumerados previamente, sobre los mensajes y las señales, se ataca proporcionando un mecanismo para almacenar los mensajes y las señales pendientes temporalmente durante el acto de la migración y luego redirigiéndolos al nuevo destino. Puede ser necesario mantener información de redireccionamiento en el sitio inicial durante algún tiempo para asegurar que los mensajes y señales pendientes se han trasladado.

Un escenario de migración. Como ejemplo representativo de auto-migración, considérese facilidades servicio disponible en el sistema operativo AIX de IBM [WALK89], que es un sistema operativo UNIX distribuido. Una funcionalidad similar dispone en el sistema operativo LOCUS [POPE85], y de hecho el sistema AIX está basado en el desarrollo de LOCUS. Este servicio se ha portado también al sistema operativo OSF/1 AD, bajo el nombre TNC [ZAJC93].

Ocurre la siguiente secuencia de eventos:

1. Cuando un proceso decide migrarse a sí mismo, selecciona una máquina objetivo y envía un mensaje de tarea remota. El mensaje lleva consigo parte de la imagen del proceso e información sobre ficheros abiertos.
2. En el sitio de destino, un proceso servidor del núcleo crea un hijo, pasándole la información del mensaje.
3. El nuevo proceso solicita datos como entorno, argumentos o información de pila a medida que la necesita para completar su operación. El código del programa se copia si está sucio o si está limpio se pagina por demanda desde el sistema de ficheros global.

4. El proceso original se señala cuando se completa la migración. Este proceso envía un mensaje final al nuevo proceso y se destruye a sí mismo.

Una secuencia similar se seguiría cuando otro proceso inicia la migración. La principal diferencia es que el proceso a migrar debe encontrarse suspendido para que pueda migrarse en un estado sin ejecución. Este procedimiento se sigue en Sprite, por ejemplo [DOUG89].

En el escenario precedente, la migración es una actividad dinámica que conlleva cierto número de pasos para trasladar la imagen del proceso. Cuando la migración se inicia por otro proceso, en vez de ser auto-migración, el otro proceso copiará la imagen del proceso y su espacio de direcciones completo en un fichero, destruirá el proceso, copiara el fichero a la otra máquina usando un servicio para la transferencia de ficheros y luego recreará el proceso desde el fichero en la máquina destino. [SMIT89] describe este enfoque.

Negociación de la migración. Otro aspecto de la migración de procesos está relacionado con las decisiones sobre la migración. En algunos casos, la decisión la toma una única entidad. Por ejemplo, si el objetivo es el equilibrado de carga, un módulo que monitoriza la carga relativa en varias máquinas y realiza la migración según sea necesario para mantener la carga equilibrada. Si se utiliza la auto-migración para permitir a un proceso acceder a funcionalidades especiales o a grandes ficheros remotos, entonces el mismo proceso puede tomar la decisión. Sin embargo, algunos sistemas permiten que el sistema destino designado participe en la decisión. Una razón para esto podría ser preservar el tiempo de respuesta de los usuarios. Un usuario en una estación de trabajo, por ejemplo, puede sufrir una degradación significativa del tiempo de respuesta si un proceso migra a su sistema, incluso si tal migración sirve para proporcionar un mejor equilibrio global.

Un ejemplo del mecanismo de negociación se encuentra en Charlotte [FINK89,ARTS89b]. La política de migración (cuándo migrar, qué proceso y a qué destino) es responsabilidad de la utilidad *Starter*, que es un proceso también responsable de la planificación a largo plazo y de la ubicación de memoria. El *Starter* puede por tanto coordinar políticas en estas tres áreas. Cada proceso *Starter* puede controlar un grupo de máquinas. El *Starter* recibe oportunamente estadísticas de carga elaboradas frecuentemente y con precisión por parte del núcleo de cada una de sus máquinas.

La decisión de migrar debe alcanzarse conjuntamente por dos procesos *Starter* (uno en el nodo de origen y uno en el nodo de destino), como se ilustra en la Figura 15.2. Suceden los siguientes pasos:

1. El *Starter* que controla el sistema origen (S) decide que un proceso P debe ser migrado a un sistema destino en concreto (D). Envía un mensaje al *Starter* de D solicitando la transferencia.
2. Si el *Starter* de D está preparado para recibir procesos, envía de vuelta un mensaje positivo de reconocimiento.
3. El *Starter* de S comunica su decisión al núcleo de S vía una llamada a un servicio (si el *Starter* ejecuta en S) o con un mensaje a cierta tarea del núcleo (KJ) de la máquina S (si el *Starter* ejecuta en otra máquina). KJ es un proceso utilizado para convertir mensajes de procesos remotos en llamadas a servicios del sistema.
4. El núcleo de S ofrece a D el envío del proceso. La oferta incluye estadísticas acerca de P, tales como su edad y cargas de procesador y comunicaciones.
5. Si D está escaso de recursos, puede rechazar la oferta. En caso contrario, el núcleo de D repite la oferta a su controlador *Starter*. La repetición incluye la misma información de la oferta de S.
6. La decisión de la política de *Starter* se comunica a D con una llamada Inmigrar.
7. D reserva los recursos necesarios para evitar el interbloqueo y problemas de control de flujo y luego envía una aceptación a S.

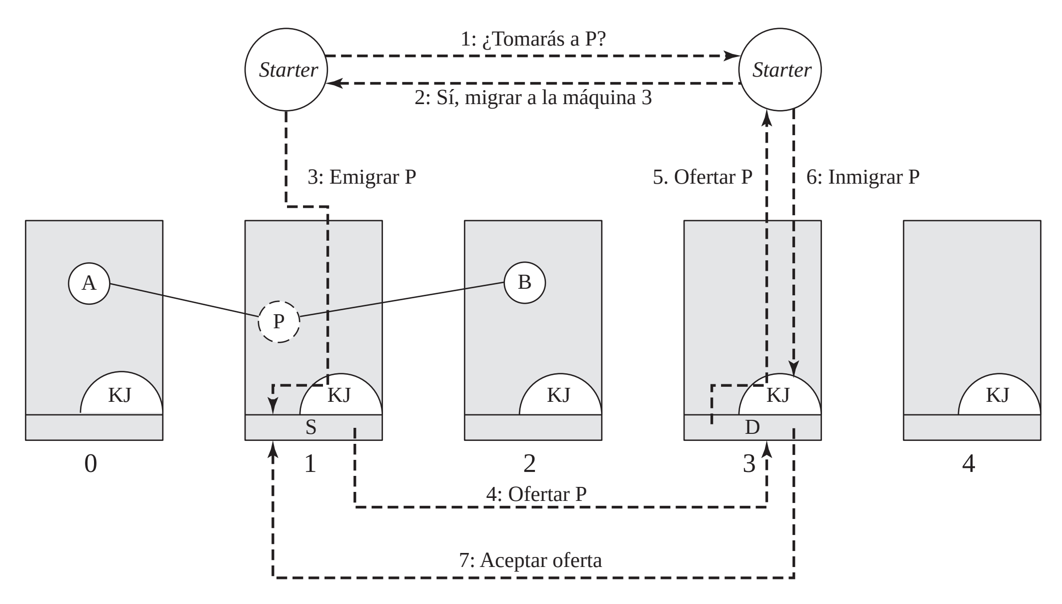


Figura 15.2. Negociación de la migración de proceso.

La Figura 15.2 también muestra otros dos procesos, A y B, que tienen enlaces abiertos con P siguiendo los pasos anteriores, la máquina 1 donde reside S, debe enviar un mensaje de actualización del enlace a ambas máquinas 0 y 2 para preservar los enlaces desde A y B a P. Los mensajes de actualización de enlace indican la nueva dirección de carga enlace mantenido por P y requieren un reconocimiento de los núcleos notificados con el objetivo de sincronización. Después de este punto un mensaje enviado a P en cualquiera de sus enlaces será enviado directamente a D. estos mensajes pueden ser intercambiados concurrentemente siguiendo los pasos que se acaban de describir. Finalmente, después del paso 7 y después de que todos los enlaces se han actualizado, S recolecta todo el contexto de P en un único mensaje y lo envía a D.

La máquina 4 también está ejecutando Charlotte pero no está involucrada en la migración y por tanto no tiene comunicación con los otros sistemas en este episodio.

DESALOJO

El mecanismo de negociación permite que el sistema de destino rechace o acepte la migración de un proceso. Además, también puede ser útil permitir que un sistema expulse a un proceso que inmigró a él. Por ejemplo, si una estación de trabajo está ociosa, uno o más procesos pueden haber migrado a ella. Cuando el usuario de la estación de trabajo pase a estar activo, puede ser necesario desalojar a los procesos inmigrados para proporcionar un adecuado tiempo de respuesta.

Un ejemplo de la capacidad de desalojo puede verse en Sprite [DOUG89]. En Sprite, que es un sistema operativo de estaciones de trabajo, cada proceso aparenta ejecutar en un único sistema durante toda su vida. Este sistema se conoce como el nodo original del proceso. Si un proceso migrar, pasa a ser un proceso foráneo en la máquina de destino. En cualquier instante la máquina de destino puede desalojar el proceso foráneo, que estará entonces obligado a volver a migrar a su nodo original.

Los elementos del mecanismo de desalojo de Sprite son como sigue:

1. Un proceso monitor en cada nodo guarda constancia de la carga actual para determinar cuándo aceptar nuevos procesos foráneos. Si el monitor detecta actividad en la consola de la estación de trabajo, iniciar un procedimiento de desalojo de cada proceso foráneo.
2. Si un proceso se expulsa, se le hace migrar de vuelta a su nodo origen. El proceso puede migrar nuevamente si hay otro nodo disponible.
3. Aunque puede llevar cierto tiempo expulsar todos los procesos, todos los procesos marcados para su desalojo se suspenden inmediatamente. Permitir a un proceso desalojado ejecutar mientras está esperando su expulsión podría reducir el tiempo durante el cual el proceso está congelado pero también reduce la potencia de proceso disponible en la máquina mientras se están realizando desalojos.
4. El espacio de direcciones completo de un proceso desalojado se transfiere a su nodo original. El tiempo de expulsar un proceso y migrarlo de vuelta a su nodo original puede ser reducido sustancialmente rescatando desde la imagen de memoria del proceso desalojado desde la máquina que lo hospedó anteriormente a medida que se referencie. Sin embargo, esto implica que la máquina que lo hospedó dedique recursos y conteste a solicitudes de servicio del proceso desalojado por un periodo de tiempo mayor del necesario.

TRANSFERENCIAS EXPULSIVAS VERSUS NO EXPULSIVAS

En la exposición de esta sección abordamos la migración de procesos expulsiva, que involucra la transferencia de procesos ejecutados parcialmente, o como poco procesos cuya creación se ha completado. Una función más sencilla es la transferencia de procesos no expulsiva, involucra sólo procesos que no han comenzado su ejecución y por tanto no requieren la transferencia del estado del proceso. En ambos tipos de transferencia, la información acerca del entorno en el cual el proceso ejecutará debe transferirse al nodo remoto. Esto puede incluir el directorio actual de trabajo del usuario, privilegios heredados por el proceso, y recursos heredados como descriptores de fichero.

La migración de procesos no expulsiva puede ser útil en el equilibrado de carga (por ejemplo, *véase* [SHIV92]). Ésta tiene la ventaja de evitar la sobrecarga de migrar procesos con todo su estado. La desventaja es que este esquema no reacciona bien a cambios súbitos en la distribución de carga.

15.2. ESTADOS GLOBALES DISTRIBUIDOS

ESTADOS GLOBALES E INSTANTÁNEAS DISTRIBUIDAS

Todos los aspectos de la concurrencia a los que nos enfrentamos en un sistema fuertemente acoplado, tales como la exclusión mutua, el interbloqueo y la inanición, también aparecen en un sistema distribuido. Las estrategias de diseño en estas áreas se complican por el hecho de que no existe un estado global del sistema. Esto es, no es posible para el sistema operativo ni para ningún proceso, conocer el estado actual de todos los procesos en un sistema distribuido. Un proceso tan sólo puede conocer el estado actual de todos los procesos en el sistema local, accediendo a los bloques de control de proceso en memoria. Para los procesos remotos, un proceso tan sólo puede conocer información de estado que se reciba vía mensajes, lo que representa el estado del proceso remoto en algún momento del pasado. Esto es análogo a la situación en astronomía: nuestro conocimiento sobre cierta estrella o galaxia distante consiste en luz y otras ondas electromagnéticas que llegan del objeto distante, y estas ondas proporcionan una imagen del objeto en algún momento del pasado. Por ejemplo, nuestro conocimiento sobre un objeto a una distancia de cinco años luz tiene cinco años de antigüedad.

Los retardos de tiempo impuestos por la naturaleza de los sistemas distribuidos complican todos los aspectos relacionados con la concurrencia. Para ilustrar esto, presentamos un ejemplo tomado de [ANDR90]. Utilizaremos gráficos proceso/evento (Figuras 15.3 y 15.4) para ilustrar el problema. En estos gráficos, hay una línea horizontal para cada proceso que representa el eje del tiempo. Un punto sobre la línea corresponde con un evento (ej., evento interno al proceso, mensaje enviado, mensaje recibido). Una caja recuadrando un punto representa una instantánea del estado local del proceso tomada en ese instante. Una flecha representa un mensaje entre dos procesos.

En nuestro ejemplo, un individuo tiene una cuenta bancaria distribuida en dos filiales de un banco. Para determinar el saldo total de la cuenta del cliente, el banco debe determinar el saldo en cada filial. Suponga que esta información debe registrarse exactamente a las 15:00 horas. La Figura 15.3a muestra una instancia en la que se encuentra un saldo de 100.00€ en la cuenta combinada. Pero la situación de la Figura 15.3b también es posible. Aquí, el saldo de la filial A está en tránsito hacia la filial B en el momento de la observación; el resultado es una lectura falsa de 0.00€. Este problema particular puede resolverse examinando todos los mensajes en tránsito en el momento de la observación. La filial A guardará registro de todas las transferencias de la cuenta, junto con la identidad del destinatario de la transferencia. Por tanto, incluiremos en el «estado» de la cuenta en la filial A tanto el saldo actual como el registro de las transferencias. Cuando se examinan dos cuentas, el observador encuentra una transferencia en curso desde la filial A destinado a la cuenta del cliente en la filial B. Dado que la cantidad todavía no ha llegado a la filial B, ésta será sumada al saldo total. Cualquier cantidad que se haya transferido y recibido será contada sólo una vez, como parte del saldo de la cuenta receptora.

Esta estrategia no es segura del todo, como muestra la Figura 15.3c. En este ejemplo, los relojes en las dos ramas no están perfectamente sincronizados. El estado de la cuenta del cliente en la filial A a las 15:00 horas indica un saldo de 100.00€. Sin embargo, esta cantidad se transfiere a continuación a la filial B a las 15:01 según el reloj de A pero llega a B a las 14:59 de acuerdo con el reloj de B. Por tanto, esa cantidad se contabiliza dos veces en la observación de las 15:00.

Para entender la dificultad con que nos encontramos y formular una solución, se definen los siguientes términos:

- **Canal.** Existe un canal entre dos procesos si intercambian mensajes. Podemos entender por canal el camino o los medios por los cuales el mensaje se transfiere. Por conveniencia, los canales se consideran unidireccionales. Así, si dos procesos intercambian mensajes, se requieren dos canales, uno por cada dirección de transferencia de mensajes.
- **Estado.** El estado de un proceso es la secuencia de mensajes que se haya enviado y recibido a través de los canales inciden en el proceso.
- **Instantánea.** Una instantánea registra el estado de un proceso. Cada instantánea incluye un registro de todos los mensajes enviados y recibidos en todos los canales desde la última instantánea.
- **Estado global.** Es el estado combinado de todos los procesos.
- **Instantánea distribuida.** Es una colección de instantáneas, una por proceso.

El problema es que el estado global real no puede determinarse debido al lapso de tiempo asociado con la transferencia de los mensajes. Podemos intentar definir un estado global recolectando instantáneas de todos los procesos. Por ejemplo, el estado global de la Figura 15.4a en el momento de tomar las instantáneas muestra un mensaje en tránsito en el canal <A,B>, uno en tránsito en el canal <A,C>, y uno en tránsito en el canal <C,A>. Los mensajes 2 y 4 están representados apropiadamente, pero el mensaje 3 no. La instantánea distribuida indica que este mensaje se ha recibido pero todavía no se ha enviado.

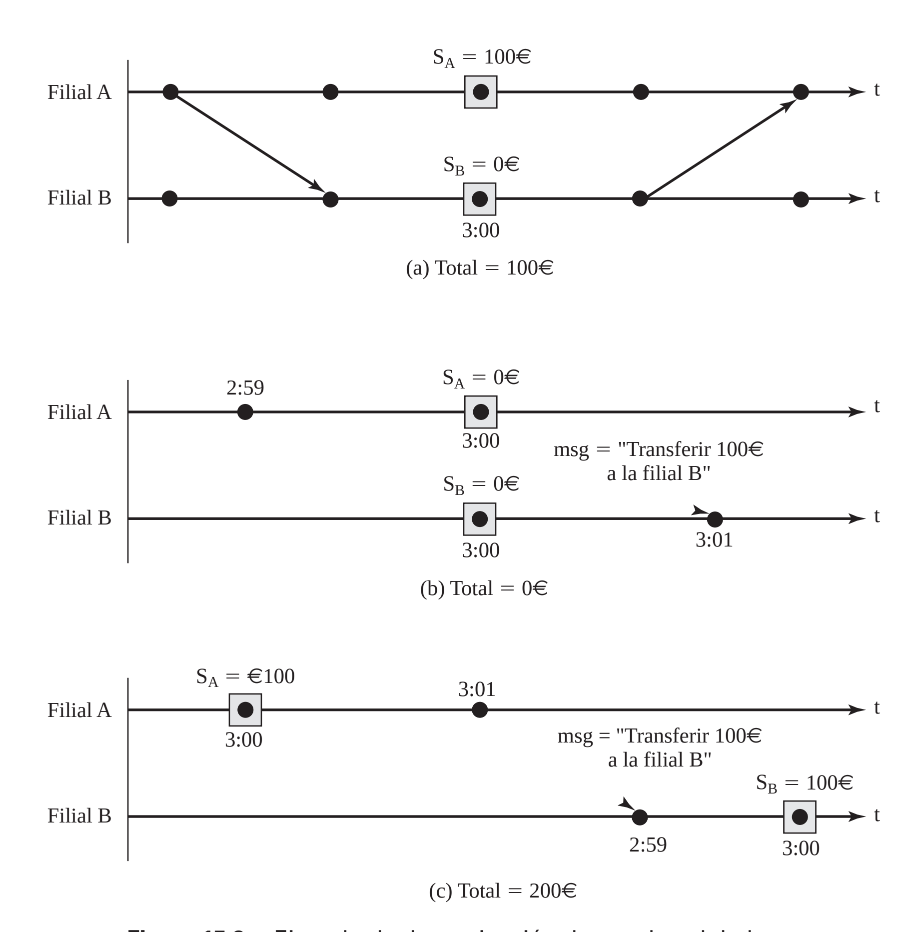


Figura 15.3. Ejemplo de determinación de estados globales.

Deseamos que la instantánea distribuida registre un estado global consistente. Un estado global es consistente si para cada estado de proceso que registra la recepción un mensaje, el envío de tal mensaje está registrado en el estado del proceso que envió dicho mensaje. La Figura 15.4b ofrece un ejemplo. Un estado global inconsistente surge si un proceso ha registrado la recepción un mensaje pero el proceso emisor correspondiente no ha registrado que el mensaje se ha enviado (Figura 15.4a).

EL ALGORITMO DE INSTANTÁNEA DISTRIBUIDA

En [CHAN85] se describe un algoritmo de instantánea distribuida que registra un estado global consistente. El algoritmo asume que los mensajes se entregan en el orden en que se envían y que no se pierden mensajes. Un protocolo de transporte fiable (ej., TCP) satisface estos requisitos. El algoritmo hace uso de un mensaje de control especial denominado **marcador**.

Algún proceso inicia el algoritmo registrando su propio estado y enviando un marcador por todos los canales salientes antes enviar ningún otro mensaje. Cada proceso *p* procede entonces como sigue. Con la primera recepción del marcador (digamos que desde el proceso *q*), el proceso receptor *p* realiza lo siguiente:

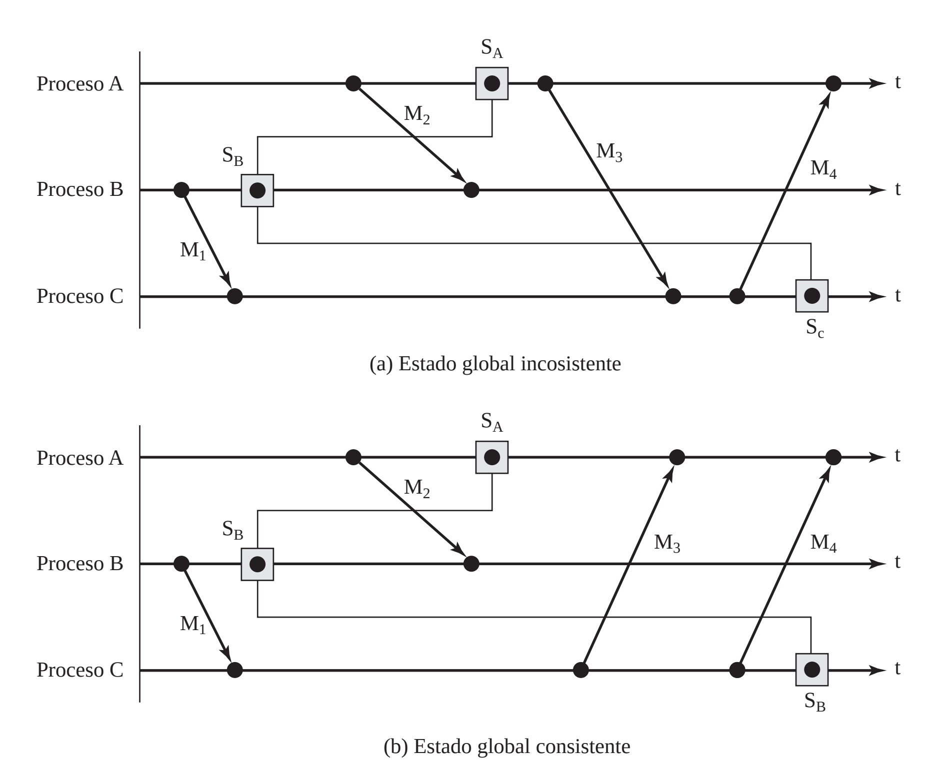


Figura 15.4. Estados globales inconsistentes y consistentes.

1. *p* registra su estado local S_p.
2. *p* registra el estado del canal entrante desde *q* a *p* como vacío.
3. *p* propaga el marcador a todos sus vecinos a través de todos los canales salientes.

Estos pasos deben realizarse atómicamente; esto es, *p* no debe enviar ni recibir mensajes hasta haber realizado los 3 pasos.

En algún momento tras recordar su estado, cuando *p* recibe un marcador desde otro canal entrante (digamos desde el proceso *r*), realizar lo siguiente:

1. *p* registra el estado del canal de *r* a *p* cómo la secuencia de mensajes que *p* ha recibido de *r* desde que *p* registró su estado local S_p en el instante en que recibía el marcador de *r*.

El algoritmo termina para un proceso una vez que ha recibido el marcador de cada canal entrante.

[ANDR90] realizan las siguientes observaciones acerca del algoritmo:

1. Cualquier proceso puede comenzar el algoritmo enviando un marcador. De hecho, varios nodos pueden independientemente decidir registrar el estado y el algoritmo, aún así, funcionará.
2. El algoritmo terminará en un tiempo finito si cada mensaje (incluyendo los mensajes marcadores) se entrega en un tiempo finito.
3. Este es un algoritmo distribuido: cada proceso es responsable de registrar su propio estado y el estado de todos los canales entrantes.

4. Una vez que se han registrado todos los estados (el algoritmo ha terminado en todos los procesos), el estado global consistente obtenido por el algoritmo puede ensamblarse en cada proceso haciendo que cada proceso envíe los datos de estado que ha registrado a través de cada canal saliente y haciendo que cada proceso reenvíe los datos de estado que recibe a través de cada canal saliente. Alternativamente, el proceso iniciador puede consultar a todos los procesos para adquirir el estado global.

5. El algoritmo no afecta y no es afectado por ningún otro algoritmo distribuido en que los procesos estén participando.

Como ejemplo del uso del algoritmo (tomado de [BEN90]), considere el conjunto de procesos ilustrado en la Figura 15.5. Cada nodo representa un proceso, y cada línea representa un canal unidireccional entre dos nodos, con la dirección indicada por una flecha. Suponga que se ejecuta el algoritmo de instantánea, con los nueve mensajes que deberán enviarse a través de cada canal saliente por cada proceso. El proceso 1 decide registrar el estado global después de enviar seis mensajes e, independientemente, el proceso 4 decide registrar el estado global tras enviar tres mensajes. Al terminar, se recolectan las instantáneas de cada proceso; los resultados se muestran en la Figura 15.6. El proceso 2 envía cuatro mensajes por cada uno de los dos canales salientes a los procesos 3 y 4 antes de registrar el estado. Ha recibido cuatro mensajes del proceso 1 antes de registrar su estado, dejando los mensajes 5 y 6 para asociarlos con el canal. El lector debe comprobar la consistencia de la instantánea: cada mensaje enviado fue bien recibido en el proceso de destino o bien registrado como en tránsito en el canal.

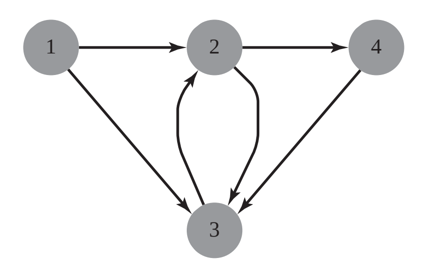


Figura 15.5. Gráfico de procesos y canales.

Proceso 1 Canales salientes 2 enviado 1, 2, 3, 4, 5, 6 3 enviado 1, 2, 3, 4, 5, 6 Canales entrantes	**Proceso 3** Canales salientes 2 enviado 1, 2, 3, 4, 5, 6, 7, 8 Canales entrantes 1 recibido 1, 2, 3 almacenado 4, 5, 6 2 recibido 1, 2, 3 almacenado 4 4 recibido 1, 2, 3
Proceso 2 Canales salientes 3 enviado 1, 2, 3, 4 4 enviado 1, 2, 3, 4 Canales entrantes 1 recibido 1, 2, 3, 4 almacenado 5, 6 3 recibido 1, 2, 3, 4, 5, 6, 7, 8	**Proceso 4** Canales salientes 3 enviado 1, 2, 3 Canales entrantes 2 recibido 1, 2 almacenado 3, 4

Figura 15.6. Un Ejemplo de una instantánea.

El algoritmo de instantánea distribuida es una herramienta potente y flexible. Puede usarse para adaptar cualquier algoritmo centralizado a un entorno distribuido, dado que la base de cualquier algoritmo centralizado es el conocimiento del estado global. Ejemplos específicos son la detección de interbloqueo o la detección de terminación de un proceso (por ejemplo, *véase* [BEN90], [LYNC96]). También puede usarse para proporcionar una instantánea de un algoritmo distribuido para permitir la recuperación si se detecta un fallo.

15.3. EXCLUSIÓN MUTUA DISTRIBUIDA

Se recuerda que en los Capítulos 5 y 6 consideramos los aspectos relacionados con la ejecución de procesos concurrentes. Los dos problemas claves que aparecieron fueron los de la exclusión mutua y el interbloqueo. Los Capítulos 5 y 6 se centraban en soluciones a este problema en el contexto de un sistema único, con uno o más procesadores pero con una memoria principal común. Al tratar con un sistema operativo distribuido y una colección de procesadores que no comparten memoria principal o reloj, aparecen nuevas dificultades y se necesitan nuevas soluciones. Los algoritmos para la exclusión mutua y el interbloqueo deben depender del intercambio de mensajes y no pueden depender del acceso a memoria común. En esta sección y en la siguiente, examinamos la exclusión mutua y el interbloqueo en el contexto de un sistema operativo distribuido.

CONCEPTOS DE EXCLUSIÓN MUTUA DISTRIBUIDA

Cuando dos o más procesos compiten por el uso de recursos del sistema, existe la necesidad de un mecanismo para hacer cumplir la exclusión mutua. Suponga que dos o más procesos requieren acceder a un recurso único no compartible, como una impresora. Durante el curso de la ejecución, cada proceso estará enviando mandatos de E/S al dispositivo, recibiendo información de estado, enviando datos o recibiendo datos. Nos referiremos a tal recurso como recurso crítico, y a la porción de programa que lo utilizan como sección crítica del programa. Es importante que sólo a un programa al tiempo se le permita estar en su sección crítica. No podemos simplemente delegar en el sistema operativo para que detecte y haga cumplir esta restricción, porque los requisitos detallados pueden no ser obvios. En el caso de una impresora, por ejemplo, deseamos que cualquier proceso individual tenga control de la impresora mientras imprime un archivo entero. De otro modo, se entremezclarían líneas de los procesos en competencia.

El éxito del uso de la concurrencia entre procesos precisa la habilidad de definir secciones críticas y de hacer cumplir la exclusión mutua. Esto es fundamental para cualquier esquema de procesamiento concurrente. Cualquier servicio o capacidad que pretenda proporcionar soporte para la exclusión mutua debe cumplir los siguientes requisitos:

1. La exclusión mutua debe hacerse cumplir: de entre todos los procesos que tienen secciones críticas para el mismo recurso u objeto compartido sólo se permite a un proceso al tiempo estar en su sección crítica.
2. Un proceso que se para en su sección no crítica debe hacerlo sin interferir con otros procesos.
3. No debe ser posible que un proceso que solicita acceso a una sección crítica sea retardado indefinidamente: ni interbloqueo ni inanición.
4. Cuando no hay proceso en una sección crítica, cualquier proceso que solicite entrar en su sección crítica deberá poder entrar en ella sin retardo.
5. No deben realizarse suposiciones sobre la velocidad relativa de los procesos o el número de procesadores.
6. Un proceso permanece dentro de su sección crítica solamente por un tiempo finito.

La Figura 15.7 muestra un modelo que podemos utilizar para examinar soluciones a la exclusión mutua en un contexto distribuido. Asumimos cierto número de sistemas interconectados por algún tipo de servicio de comunicaciones. Dentro de cada sistema, asumimos que alguna función o proceso dentro del sistema operativo es responsable de la ubicación de recursos. Cada uno de esos procesos controla un número de recursos y sirve para un número de procesos de usuario. La tarea es encontrar un algoritmo mediante el cual estos procesos puedan cooperar para conseguir la exclusión mutua.

Los algoritmos para la exclusión mutua pueden ser centralizados o distribuidos. En un **algoritmo centralizado** del todo, se designa un nodo como nodo de control para controlar el acceso a todos los objetos compartidos. Cuando cualquier proceso requiere acceso a un recurso crítico, emite una solicitud a su proceso local controlador de recursos. Este proceso, a su vez, envía un mensaje de solicitud al nodo de control, que devuelve un mensaje de respuesta (permiso) cuando el objeto compartido pasa a estar disponible. Cuando un proceso ha terminado con un recurso, envía un mensaje de liberación al nodo de control. Tal algoritmo centralizado tiene dos propiedades clave:

1. Solamente el nodo de control toma decisiones de ubicación de recursos.
2. Toda la información necesaria está concentrada en el nodo de control, incluyendo la identidad y localización de todos los recursos y el estado de ubicación de cada recurso.

La solución centralizada es directa, y es fácil ver cómo puede conseguirse la exclusión mutua: el nodo de control no satisfará la solicitud de un recurso hasta que el recurso se haya liberado. Sin embargo, un esquema como éste sufre ciertos inconvenientes. Si el nodo de control falla, entonces se rompe el mecanismo de exclusión mutua, por lo menos temporalmente. Es más, cada ubicación y liberación de recursos precisa el intercambio de mensajes con el nodo de control. Así, el nodo de control puede ser un cuello de botella.

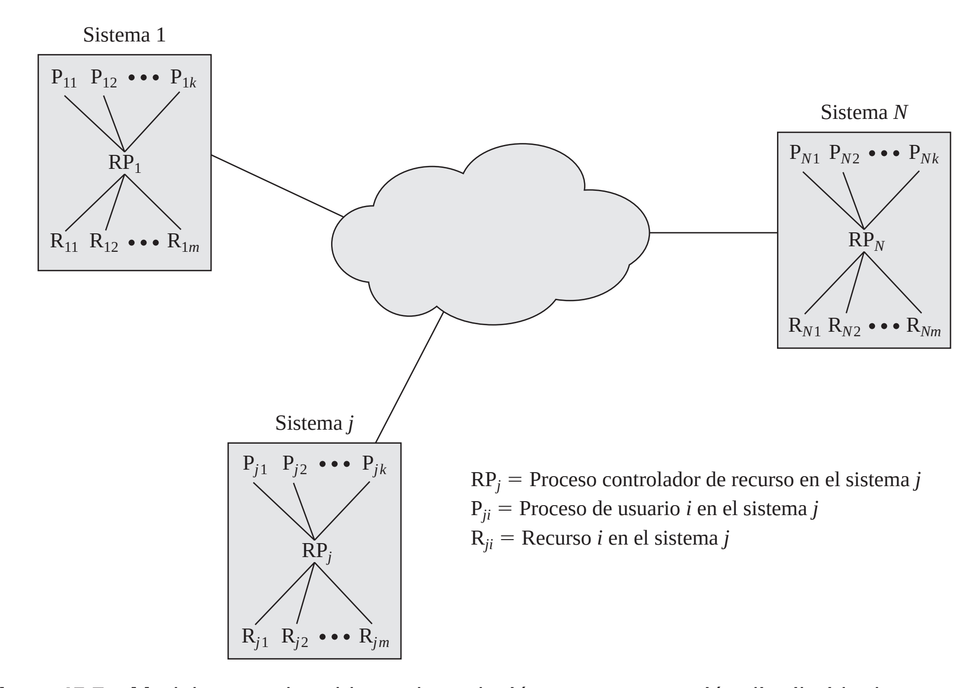


Figura 15.7. Modelo para el problema de exclusión mutua en gestión distribuida de procesos.

Debido a los problemas del algoritmo centralizado, ha habido interés en el desarrollo de algoritmos distribuidos. Un **algoritmo distribuido** del todo se caracteriza por las siguientes propiedades [MAEK87]:

1. Todos los nodos tienen igual cantidad de información, por término medio.
2. Cada nodo tiene solamente una visión parcial del sistema total y debe tomar decisiones basándose en esa información.
3. Todos los nodos tienen igual responsabilidad sobre la decisión final.
4. Todos los nodos realizan un esfuerzo similar en la toma de la decisión final.
5. El fallo de un nodo, en general, no conlleva el colapso total del sistema.
6. No existe un reloj global común en el sistema con el cual regular los eventos de tiempo.

Los puntos 2 y 6 pueden necesitar cierta elaboración. Con respecto al punto 2, algunos algoritmos distribuidos necesitan que toda la información conocida por cualquier nodo sea comunicada a todos los otros nodos. Incluso en este caso, en cualquier momento dado, alguna información estará en tránsito y no habrá llegado a todos los otros nodos. Así, debido a los retardos de tiempo en la comunicación de mensajes, la información de un nodo normalmente no está completamente actualizada y es en este sentido que es solamente información parcial.

Con respecto al punto 6, debido al retardo en la comunicación entre sistemas, es imposible mantener un reloj de ámbito global que sea instantáneamente disponible en todos los sistemas. Es más, también es técnicamente impracticable mantener un reloj central y que todos los relojes locales se sincronicen precisamente con ese reloj central; a lo largo de un periodo de tiempo, habrá cierto desfase entre los varios relojes locales que causará una pérdida de sincronización.

Es el retardo de las comunicaciones, junto con la falta de un reloj común, lo que hace mucho más difícil desarrollar mecanismos de exclusión mutua en un sistema distribuido en comparación con un sistema centralizado. Antes del ver algunos algoritmos para la exclusión mutua distribuida, examinamos un enfoque común para superar el problema de la inconsistencia de relojes.

ORDENACIÓN DE EVENTOS EN UN SISTEMA DISTRIBUIDO

La ordenación temporal de eventos es fundamental para la operación de la mayoría de algoritmos distribuidos para la exclusión mutua y el interbloqueo. La falta de un reloj común o los medios para la sincronización de relojes locales es también una restricción importante. El problema puede expresarse de la siguiente manera. Queremos poder ser capaces de decir que un evento *a* en un sistema *i* sucede antes (o después) del evento *b* en el sistema *j*, y querríamos ser capaces de alcanzar consistentemente esta conclusión en todos los sistemas de la red. Desafortunadamente, esta sentencia no es precisa por dos razones. Primero, debe haber un retardo entre la ocurrencia real de un evento y el momento en que es observado en algún otro sistema. Segundo, la falta de sincronización lleva a una variación en las lecturas del reloj en diferentes sistemas.

Para superar estas dificultades, un método conocido como sellos de tiempo fue propuesto por Lamport [LAMP87], que ordena eventos en un sistema distribuido sin utilizar relojes físicos. Esta técnica es tan eficiente y efectiva que se utiliza en la gran mayoría de algoritmos distribuidos para la exclusión mutua y el interbloqueo.

Para comenzar, necesitamos decidir una definición del término *evento*. En última instancia, lo que nos interesa son acciones que ocurren en un sistema local, como que un proceso entra o abandona su sección crítica. Sin embargo, en un sistema distribuido, la manera en que los procesos interactúan

es por medio de mensajes. Por tanto, tiene sentido asociar eventos con mensajes. Un evento local puede vincularse a un mensaje de manera muy sencilla, por ejemplo, un proceso puede enviar un mensaje cuando desea entrar en su sección crítica o cuando está abandonando su sección crítica. Para evitar la ambigüedad, asociamos eventos solamente con el envío de mensajes, no con la recepción de mensajes. Así, cada vez que un proceso transmite un mensaje, se define un evento que corresponde con el instante en que el mensaje abandona el proceso.

El esquema de sellos de tiempo pretende ordenar los eventos consistentes en la transmisión de mensajes. Cada sistema i en la red mantiene un contador local, C_i, que funciona como un reloj. Cada vez que un sistema transmite un mensaje, primero incrementa su reloj en 1. El mensaje envía con la forma

$$(m, T_i, i)$$

donde

m = contenidos del mensaje

T_i = sello de tiempo para este mensaje, establecido igual a C_i

i = identificador numérico de este sitio

Cuando se recibe un mensaje, el sistema receptor j pone su reloj a uno más que el máximo entre su valor actual y el sello de tiempo entrante:

$$C_j \leftarrow 1 + \max[C_j , T_i]$$

En cada sitio, la ordenación de los eventos viene determinada por las siguientes reglas. Para un mensaje x de un sitio i y un mensaje y de un sitio j, se dice que x precede a y si se cumple una de las siguientes condiciones:

1. Si $T_i < T_j$, o
2. Si $T_i = T_j$ e $i < j$.

El tiempo asociado con cada mensaje es el sello de tiempo que acompaña al mensaje, y la ordenación de estos tiempos viene determinada por las dos reglas precedentes. Esto es, dos mensajes con el mismo sello de tiempo se ordenan por los números de sus sitios. Dado que la aplicación de estas reglas es independiente del sitio, esta solución impide cualquier problema de deriva entre los diversos relojes de los procesos en comunicación.

En la Figura 15.8 se muestra un ejemplo de la operación de este algoritmo. Hay tres sitios, cada uno de los cuales está representado por un proceso que controla el algoritmo de sello de tiempo. El proceso P_1 comienza con un valor de reloj de 0. Para transmitir el mensaje *a*, incrementa su reloj en 1 y transmite (*a*,1,1), donde el primer valor numérico es el sello de tiempo y el segundo es la identidad del sitio. Este mensaje se recibe por los procesos en los sitios 2 y 3. En ambos casos, el reloj local tiene un valor de 0 y es puesto a un valor de 2 = 1 + max[0,1]. P_2 emite el siguiente mensaje, incrementando primero su reloj a 3. Con la recepción de este mensaje, P_1 y P_3 incrementan sus relojes a 4. Entonces P_1 emite el mensaje *b* y P_3 emite el mensaje *j* aproximadamente a la vez y con el mismo sello de tiempo. Debido al principio de ordenación visto previamente, esto no causa confusión. Después de que todos estos eventos han tenido lugar, el orden de los mensajes es el mismo en todos los sitios, esto es $\{a,x,b,j\}$.

El algoritmo funciona a pesar de las diferencias en los tiempos de transmisión entre parejas de sistemas, tal como ilustra la Figura 15.9. Aquí, P_1 y P_4 emite en mensajes con el mismo sello de tiem-

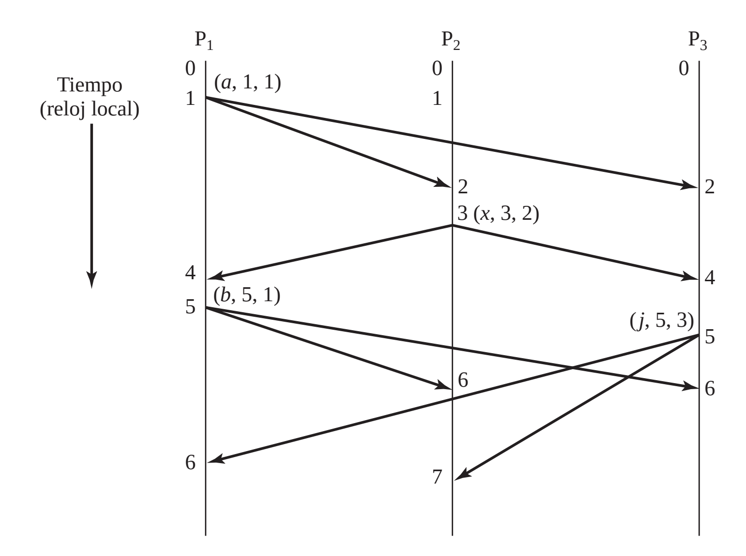


Figura 15.8. Ejemplo de operación del algoritmo de sellado de tiempo.

po. El mensaje desde P_1 llega más temprano que el de P_4 al sitio 2 pero después que el de P_4 al sitio 3. Sin embargo, después de que todos los mensajes se hayan recibido en todos los sitios, el orden de los mensajes es el mismo en todos ellos: $\{a,q\}$.

Obsérvese que la ordenación impuesta por este esquema no se corresponde necesariamente a la secuencia real de tiempo. Para los algoritmos basados en el esquema de sellado de tiempo, no es importante qué evento realmente sucedió primero. Tan sólo es importante que todos los procesos que implementan el algoritmo acuerden un orden único que es impuesto sobre los eventos.

En los dos ejemplos que se acaban de discutir, cada mensaje se envía de un proceso a todos los demás procesos. Si algún mensaje no se envía de este modo, algunos sitios no reciben todos los mensajes en el sistema y por lo tanto es imposible que todos los sitios tengan la misma ordenación de mensajes. En tal caso, existirá un conjunto de ordenaciones parciales. Sin embargo, nuestra preocupación principal es el uso de sellos de tiempo en algoritmos distribuidos para la exclusión mutua y la detección de interbloqueo. En tales algoritmos, un proceso normalmente envía un mensaje (con su sello de tiempo) a todos los demás procesos, y los sellos de tiempo se utilizan para determinar cómo procesar los mensajes.

COLA DISTRIBUIDA

Primera versión. Uno de los primeros enfoques propuestos para proporcionar exclusión mutua distribuida está basado en el concepto de cola distribuida [LAMP78]. El algoritmo se basa en las siguientes asunciones:

1. Un sistema distribuido consiste en N nodos, numerados de manera única del 1 al N. Cada nodo contiene un proceso que realiza las peticiones de acceso en exclusión mutua a los recursos en nombre de otros procesos; este proceso también sirve de árbitro para resolver las solicitudes entrantes de otros nodos que se solapen en el tiempo.
2. Los mensajes enviados de un nodo a otro se reciben en el mismo orden en que se enviaron.
3. Cada mensaje se entrega correctamente a su destinatario en un tiempo finito.

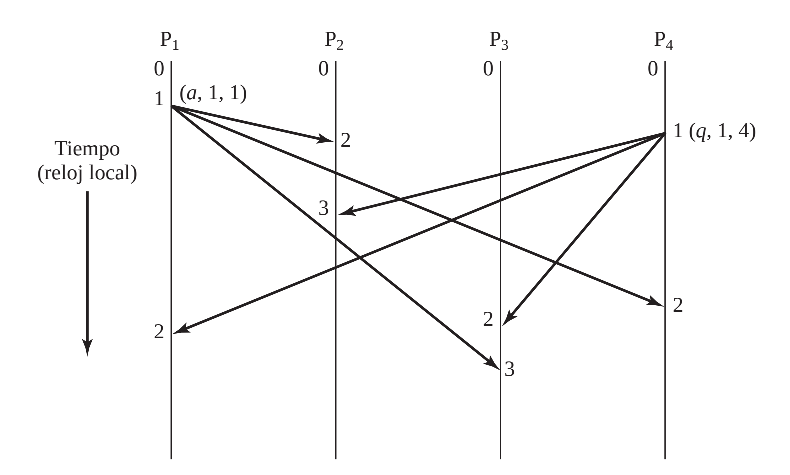


Figura 15.9. Otro ejemplo de operación del algoritmo de sellado de tiempo.

4. La red está completamente conectada; esto significa que cualquier proceso puede enviar mensajes directamente a cualquier otro proceso, sin requerir la intermediación de otro proceso que retransmita el mensaje.

Las suposiciones 2 y 3 pueden conseguirse mediante el uso de un protocolo de transporte fiable, como TCP (Capítulo 13).

Por simplicidad, se describe el algoritmo para el caso en que cada sitio controla sólo un único recurso. La generalización a múltiples recursos es trivial.

El algoritmo intenta generalizar un algoritmo que funcionaría de manera inmediata en un sistema centralizado. Si un proceso central único gestiona el recurso, puede encolar las solicitudes entrantes y darles acceso en estricto orden FIFO. Para conseguir este mismo algoritmo en un sistema distribuido, todos los sitios deben tener una copia de la misma cola. Puede utilizarse el sellado de tiempo para conseguir que todos los nodos se pongan de acuerdo en el orden en que deben concederse las solicitudes de recursos. Aparece una complicación: dada la cantidad finita de tiempo que le lleva a un mensaje atravesar la red, existe el peligro de que dos sitios distintos no se pongan de acuerdo en qué proceso está en cabeza de la cola. Considere la Figura 15.9. Hay un punto en el que el mensaje *a* ha llegado a P_2, y el mensaje *q* ha llegado a P_3, pero ambos mensajes están todavía en tránsito hacia otros procesos. Así, existe un periodo de tiempo en que P_1 y P_2 consideran que el mensaje *a* está en cabeza de la cola y P_3 y P_4 consideran que el mensaje *q* está en cabeza de la cola. Esto puede dar lugar a la violación del requisito de exclusión mutua. Para evitar esto, se impone la siguiente regla: para que un proceso pueda tomar una decisión basándose en su propia cola, necesita haber recibido un mensaje de cada uno de los otros sitios de manera que pueda garantizar que no hay ningún mensaje en tránsito más reciente que el cabeza de su cola. Esta regla se explica en la parte 3b del algoritmo descrito a continuación.

En cada sitio se mantiene una estructura de datos que guarda constancia del mensaje recibido más recientemente de cada sitio (incluyendo el mensaje más reciente generado en el propio sitio). Lamport se refiere a esta estructura como cola; realmente es un vector con una entrada por cada sitio. En cualquier momento, la entrada $q[j]$ en el vector local contiene un mensaje de P_j. El vector se inicializa como sigue:

$$q[j] = (\text{Liberar}, 0, j) \qquad j = 1, ..., N$$

En este algoritmo se utilizan tres tipos de mensajes:

- (Solicitud, T_i, i). Una solicitud de acceso a un recurso realizada por P_i.
- (Respuesta, T_j, j). P_j concede acceso a un recurso bajo su control.
- (Liberar, T_k, k). P_k libera un recurso que previamente se le concedió.

El algoritmo es como sigue:

1. Cuando P_i necesita acceder a un recurso, emite una solicitud (Solicitud, T_i, i), sellada con el tiempo del valor actual de su reloj local. Pone el mensaje en su propio vector local en $q[i]$ y envía el mensaje a todos los demás procesos.
2. Cuando P_j recibe (Solicitud, T_i, i), pone el mensaje en su propio vector en $q[i]$. Si $q[i]$ no contiene un mensaje de solicitud, P_j transmite (Respuesta, T_j, j) a P_i. Ésta es la acción que implementa la regla descrita anteriormente, que asegura que no hay mensaje de Solicitud en tránsito en el momento de la decisión.
3. P_i puede acceder el recurso (entrar en su sección crítica) cuando se cumplan las siguientes dos condiciones:
   a) La propia Solicitud de P_i en el vector q es el mensaje de Solicitud más reciente del vector; como los mensajes se ordenan de manera consistente en todos los sitios, esta regla permite que uno y sólo uno de los procesos acceda al recurso en un momento dado.
   b) Todos los demás mensajes en el vector local son posteriores al mensaje en $q[i]$; esta regla garantiza que P_i conoce acerca de todas las solicitudes que preceden a su solicitud actual.
4. P_i libera un recurso emitiendo un mensaje (Liberar, T_i, i), que pone en su propio vector y transmite a todos los demás procesos.
5. Cuando P_i recibe (Liberar, T_j, j), reemplaza los contenidos actuales de $q[j]$ con este mensaje.
6. Cuando P_i recibe (Respuesta, T_j, j), reemplaza los contenidos actuales de $q[j]$ con este mensaje.

Se ve claramente que este algoritmo consigue la exclusión mutua, es equitativo, evita el interbloqueo y la hambruna:

- **Exclusión mutua.** Las solicitudes de entrada en la sección crítica se sirven conforme a la ordenación impuesta por el mecanismo de sello de tiempo. Una vez que P_i decide entrar en su sección crítica, no puede haber en el sistema ningún otro mensaje de Solicitud transmitido antes que el suyo propio. Esto es cierto porque, por entonces, P_i ha recibido necesariamente un mensaje de cada uno de los otros sitios y cada uno de los otros mensajes son anteriores a su propio mensaje Solicitud. Podemos estar seguros de esto gracias al mecanismo del mensaje de Respuesta; recuerde que los mensajes entre dos sitios no pueden llegar fuera de orden.
- **Equitativo.** Las solicitudes se conceden estrictamente sobre la base de la ordenación de sellos de tiempo. Por lo tanto, todos los procesos tiene igual oportunidad.
- **Libre de interbloqueo.** Dado que la ordenación por sellos de tiempo se mantiene consistentemente en todos los sitios, no pueden ocurrir interbloqueos.
- **Libre de hambruna.** Una vez que P_i ha completado su sección crítica, transmite un mensaje Liberar. Esto tiene el efecto de borrar el mensaje de Solicitud de P_i en todos los otros sitios, permitiendo que algún otro proceso entre en su sección crítica.

Como medida de la eficiencia de este algoritmo, nótese que para garantizar la exclusión, se necesitan $3 \times (N - 1)$ mensajes: $(N - 1)$ mensajes de Solicitud, $(N - 1)$ mensajes de Respuesta y $(N - 1)$ mensajes de Liberar.

Segunda versión. En [RICA81] se propuso una mejora del algoritmo de Lamport. Se busca optimizar el algoritmo original eliminando los mensajes de Liberar. Imperan las mismas restricciones que antes, excepto que no es necesario que todos los mensajes enviados desde un proceso a otro sean recibidos en el mismo orden en el que fueron enviados.

Como antes, cada sitio contiene un proceso que controla la ubicación de recursos. Este proceso mantiene un array *q* que obedece a las siguientes reglas:

1. Cuando P_i necesita acceder a un recurso, emite una solicitud (Solicitud, T_i, *i*), sellada con el tiempo del valor actual de su reloj local. Pone el mensaje en su propio vector local en *q*[*i*] y envía el mensaje a todos los demás procesos.
2. Cuando P_j recibe (Solicitud, T_i, *i*), obedece a las siguientes reglas:
   a) Si P_j está actualmente en su sección crítica, aplaza el envío del mensaje Respuesta (*véase* la siguiente Regla 4).
   b) Si P_j no está esperando para entrar en su sección crítica (no ha emitido una Solicitud que está todavía pendiente), transmite (Respuesta, T_j, *j*) a P_i.
   c) Si P_j está esperando para entrar en su sección crítica y si el mensaje entrante sigue a la solicitud de P_j, entonces coloca este mensaje en su propio array en *q*[*i*] y aplaza el envío del mensaje Respuesta.
   d) Si P_j está esperando para entrar en su sección crítica y si el mensaje entrante precede a la solicitud de P_j, entonces coloca este mensaje en su propio array en *q*[*i*] y transmite el mensaje (Respuesta, T_j, *j*) a P_i.
3. P_i puede acceder al recurso (entrar en su sección crítica) cuando ha recibido un mensaje Respuesta de todos los demás procesos.
4. Cuando P_i abandona su sección crítica, libera el recurso enviando un mensaje Respuesta a cada Solicitud pendiente.

El diagrama de transiciones de estado para cada proceso se muestra en la Figura 15.10.

Para resumir, cuando un proceso desea entrar en su sección crítica, envía un mensaje de Solicitud con sello de tiempo a todos los demás procesos. Cuando recibe una Respuesta de todos los otros procesos, puede entrar en su sección crítica. Cuando un proceso recibe una Solicitud de otro proceso, debe eventualmente enviar la correspondiente Respuesta. Si un proceso no desea entrar en su sección crítica, envía la Respuesta en ese momento. Si desea entrar en su sección crítica, compara el sello de tiempo de su Solicitud con el de la última Solicitud recibida, y si el último es más reciente, aplaza su Respuesta; en otro caso la Respuesta se envía en ese momento.

Con este método, se requieren $2 \times (N - 1)$ mensajes: $(N - 1)$ mensajes de Solicitud para que P_i indique su intención de entrar en su sección crítica, y $(N - 1)$ mensajes de Respuesta para permitir el acceso que se ha solicitado.

El uso de sellos de tiempo en este algoritmo asegura la exclusión mutua. También evita el interbloqueo. Para aprobar esto último, asúmase lo opuesto: que es posible que, cuando no hay más mensajes en tránsito, tenemos una situación en la cual cada proceso ha transmitido una Solicitud y no ha recibido la necesaria Respuesta. Esta situación no puede suceder, dado que la decisión de diferir una Respuesta se basa en una relación que ordena las Solicitudes. Por tanto hay una Solicitud que tiene el sello de tiempo más antiguo y que recibirá todas las Respuestas necesarias. El interbloqueo es por lo tanto imposible.

La inanición también se evita porque las Solicitudes están ordenadas. Dado que las Solicitudes se sirven en dicho orden, cada Solicitud llega a ser en algún momento la más antigua y se servirá entonces.

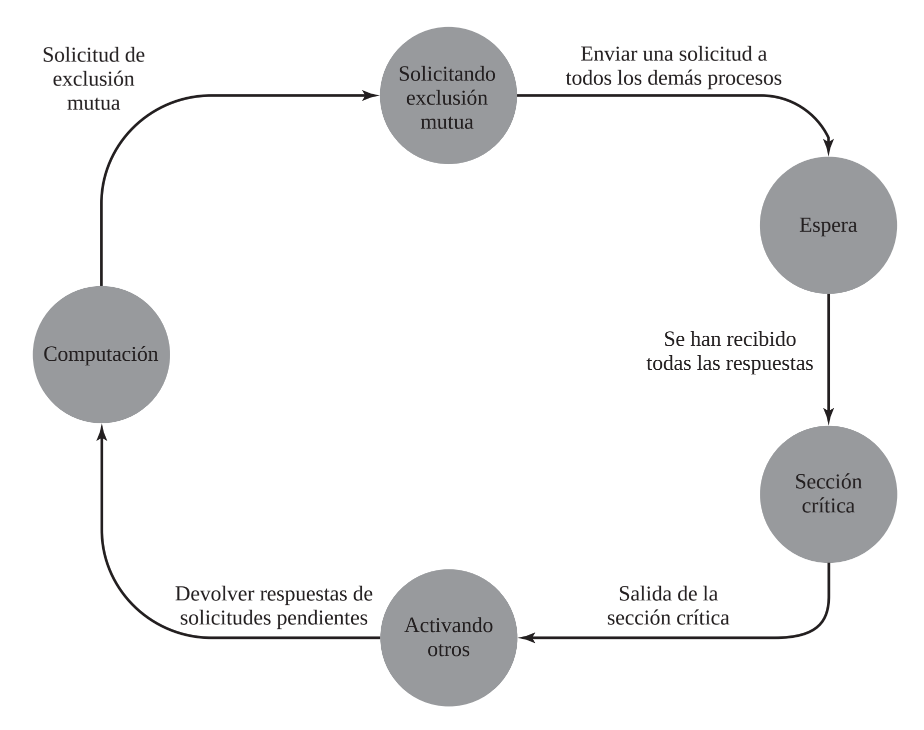


Figura 15.10. Diagrama de estados para el algoritmo en [RICA81].

UN ENFOQUE DE PASO DE TESTIGO

Cierto número de investigadores han propuesto un enfoque bastante diferente para la exclusión mutua, que involucra el paso de un testigo entre los procesos participantes. El testigo es una entidad que en cualquier momento posee uno de los procesos. El proceso que tiene el testigo puede entrar en su sección crítica sin pedir permiso. Cuando un proceso abandona su sección crítica, pasa el testigo a otro proceso.

En esta subsección, vemos uno de los más eficientes de estos esquemas. Fue propuesto por primera vez en [SUZU82]; una propuesta conceptualmente equivalente aparece también en [RICA83]. Para este algoritmo se necesitan dos estructuras de datos. El testigo, que se pasa de un proceso a otro, es realmente un vector, testigo, cuya entrada *k* registra el sello de tiempo de la última vez que el testigo visitó al proceso P_k. Además, cada proceso mantiene un vector, solicitud, cuya entrada *j* registra el sello de tiempo de la última Solicitud recibida de P_j.

El procedimiento es como sigue. Inicialmente, el testigo se le asigna arbitrariamente a uno de los procesos. Cuando un proceso desea utilizar su sección crítica, puede hacerlo si está actualmente en posesión del testigo; en otro caso difunde un mensaje de solicitud con sello de tiempo a todos los demás procesos y espera hasta que recibe el testigo. Cuando el proceso P_j abandona su sección crítica, debe transmitir el testigo a algún otro proceso. Escoge el siguiente proceso que va a recibir el testigo buscando en el vector solicitud en el orden $j+1$, $j+2$, …, 1, 2, …, $j-1$ la primera entrada [*k*] tal que el sello de tiempo para la última solicitud de P_k por el testigo es mayor que el valor registrado en el testigo para la última vez que P_k tuvo el testigo; esto es, solicitud [*k*] > testigo [*k*].

La Figura 15.11, muestra el algoritmo, que tiene dos partes. La primera parte trata del uso de la sección crítica y consiste en un preámbulo, seguido de la sección crítica y seguida por un epílogo. La

```
if (!testigo_presente)
{
      reloj++;                                          /* Preámbulo */
      difundir(solicitud, reloj, i);
      esperar(acceso, testigo);
      testigo_presente = cierto;
}
testigo_tomado = cierto;
<sección crítica>;
testigo[i] = reloj;                                     /* Epílogo */
testigo_tomado = falso;
for (int j = i + 1; j < n; j++)
{
      if (solicitud(j) > testigo[j] && testigo_presente)
      {
            testigo_presente = falso;
            enviar(acceso, testigo[j]);
      }
}
```

(a) Primera parte

```
if (recibir(solicitud, k, j))
{
      solicitud (j) = max(solicitud(j), k);
      if (testigo_presente && !testigo_tomado)
            <texto del epílogo>;
}
```

(b) Segunda parte

Notación

enviar(j, acceso, testigo)	envía un mensaje de tipo acceso, con testigo del proceso j
difundir(solicitud, reloj, i)	envía mensaje del proceso i de tipo solicitud, con sello de tiempo reloj, a todos los demás procesos
recibir(solicitud, t, j)	recibe mensaje del proceso j de tipo solicitud, con sello de tiempo t

Figura 15.11. Algoritmo de paso de testigo (para el Proceso P_i).

segunda parte tiene que ver con la acción a tomar cuando se recibe una solicitud. La variable reloj es el contador local utilizado para la función de sello de tiempo. La operación esperar (acceso, testigo) hace que el proceso espere hasta que se recibe un mensaje del tipo «acceso», que entonces se coloca en el vector testigo.

El algoritmo requiere:

- *N* mensajes (*N* – 1 para difundir la solicitud y 1 para transferir el testigo) cuando el proceso solicitante no tiene el testigo.
- Ningún mensaje, si el proceso ya tiene el testigo.

15.4. INTERBLOQUEO DISTRIBUIDO

En el Capítulo 6, definimos interbloqueo como el bloqueo permanente de un conjunto de procesos que bien compiten por recursos del sistema o bien se comunican entre sí. Esta definición es válida para un sistema único tanto como para un sistema distribuido. Como con la exclusión mutua, el interbloqueo presenta problemas más complejos en un sistema distribuido, comparado con un sistema con memoria compartida. El manejo del interbloqueo se complica en un sistema distribuido porque ningún nodo tiene conocimiento preciso del estado actual del sistema global y porque la transferencia de cada mensaje entre procesos involucra un retardo impredecible.

La literatura ha prestado atención a dos tipos de interbloqueo distribuido: aquellos que surgen en la ubicación de recursos, y aquellos que surgen con la comunicación de mensajes. En los interbloqueos por recursos, los procesos intentan acceder a recursos, tales como objetos en una base de datos o recursos de E/S en un servidor; el interbloqueo sucede si cada proceso de un conjunto de procesos solicita un recurso que tiene otro proceso del conjunto. En los interbloqueos en comunicaciones, los mensajes son los recursos por los cuales esperan los procesos; el interbloqueo sucede si cada proceso de un conjunto está esperando un mensaje de otro proceso en el conjunto y ningún proceso en el conjunto envía nunca un mensaje.

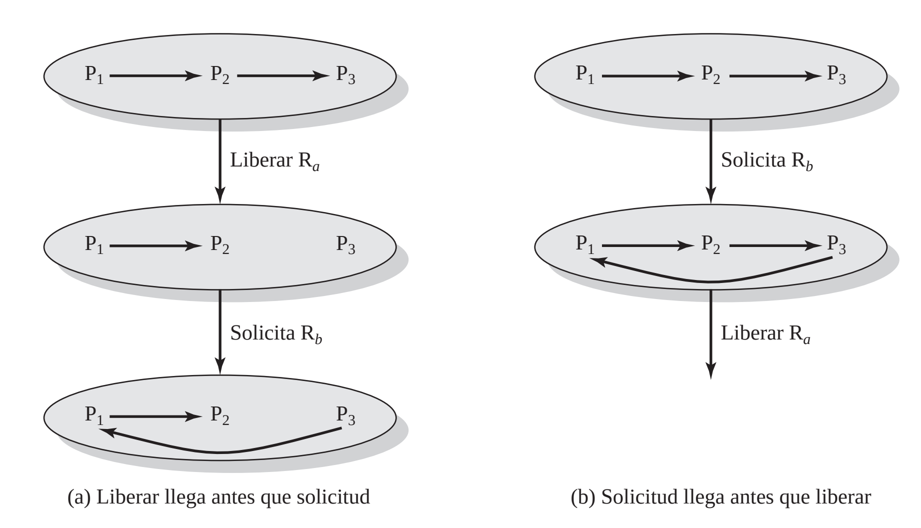
INTERBLOQUEO EN LA UBICACIÓN DE RECURSOS

Recuerde del Capítulo 6, el interbloqueo en la ubicación de recursos sólo existe si se cumplen todas las siguientes condiciones:

- **Exclusión mutua.** Sólo un proceso puede utilizar un recurso en un momento dado. Ningún proceso puede acceder a un recurso que se ha reservado para otro proceso.
- **Tener y esperar.** Un proceso puede tener reservados recursos mientras espera la asignación de otros.
- **No expulsión.** No se puede quitar a la fuerza ningún recurso al proceso que lo tiene.
- **Espera circular.** Existe una cadena cerrada de procesos, tal que cada proceso tiene al menos un recurso requerido por el siguiente proceso de la cadena.

El propósito de un algoritmo que trate con el interbloqueo es tanto prevenir la formación de la espera circular como detectar su ocurrencia real o potencial. En un sistema distribuido, los recursos están distribuidos sobre varios sitios y el acceso a ellos se regula por procesos de control que no tienen un conocimiento completo y actualizado del estado global del sistema y por tanto deben tomar sus decisiones sobre la base de información local. Por tanto, se necesitan nuevos algoritmos de interbloqueo.

Un ejemplo de las dificultades que se encuentran en la gestión distribuida de interbloqueos es el fenómeno del interbloqueo fantasma. Un ejemplo del interbloqueo fantasma se muestra en la Figura 15.12. La notación $P_1 \rightarrow P_2 \rightarrow P_3$ significa que P_1 está parado esperando por un recurso que tiene P_2,



(a) Liberar llega antes que solicitud

(b) Solicitud llega antes que liberar

Figura 15.12. Interbloqueo fantasma.

y P_2 está esperando por un recurso que tiene P_3. Digamos que al principio del ejemplo, P_3 posee el recurso R_a y P_1 posee el recurso R_b. Suponga ahora que P_3 emite primero un mensaje liberando R_a y luego un mensaje solicitando R_b. Si el primer mensaje alcanza al proceso de detección de ciclo antes que el segundo, sucede la secuencia de la Figura 15.12a, donde se reflejan adecuadamente las necesidades de recursos. Si, sin embargo, el segundo mensaje llega antes que el primer mensaje, se detectará un interbloqueo (Figura 15.12b). Esta es una detección falsa, no un interbloqueo real, debido a la falta de un estado global como el que existiría en un sistema centralizado.

Prevención de interbloqueo. Dos de las técnicas de prevención de interbloqueo expuestas en el Capítulo 6 pueden usarse en un entorno distribuido.

1. La condición de espera circular puede prevenirse definiendo una ordenación lineal de los tipos de recurso. Si un proceso ha ubicado recursos del tipo R, entonces sólo puede solicitar a continuación recursos de tipos que sigan a R en la ordenación. Una desventaja importante de este método es que los recursos pueden no solicitarse en el orden en el cual se van a utilizar; así los recursos pueden estar reservados más tiempo del necesario.
2. La condición de tener-y-esperar puede prevenirse exigiendo que un proceso solicite a la vez todos los recursos que necesita y bloqueando al proceso hasta que todas las solicitudes puedan satisfacerse simultáneamente. Este enfoque es ineficiente por los siguientes motivos. Primero, un proceso puede quedar esperando durante mucho tiempo hasta que se complete la solicitud por todos sus recursos, cuando de hecho podría haber avanzado teniendo sólo una parte de los recursos. Segundo, los recursos asignados a un proceso pueden permanecer sin usarse durante un periodo considerable, durante el cual le son denegados a otros procesos.

Ambos métodos requieren que el proceso determine sus necesidades de recursos con antelación. Este no es siempre el caso; un ejemplo es una aplicación de base de datos en la cual pueden añadirse nuevos elementos dinámicamente. Como ejemplo de una solución que no necesita este conocimiento anticipado, consideramos dos algoritmos propuestos en [ROSE78]. Estos se desarrollaron en el contexto de trabajo de base de datos, así que debemos hablar de transacciones en vez de procesos.

Los métodos propuestos hacen uso de sellos de tiempo. Cada transacción acarrea durante su tiempo de vida el sello de tiempo de su creación. Esto establece una ordenación estricta de las transaccio-

nes. Si un recurso R que ya está siendo utilizado por la transacción T1 se solicita por parte de otra transacción T2, el conflicto se resuelve comparando sus sellos de tiempo. Esta comparación se utiliza para prevenir la formación de una condición de espera circular. Los autores proponen dos variaciones a este método básico, referidas como método «esperar-morir» (*wait-die*) y método «herir-esperar» (*wound-wait*).

Supongamos que actualmente T1 tiene R y que T2 realiza una solicitud. Para el método **esperar-morir**, la Figura 15.13a muestra el algoritmo utilizado por el gestor de recurso en el sitio de R. Los sellos de tiempo de las dos transacciones se denotan como e(T1) y e(T2). Si T2 es más viejo, se bloquea hasta que T1 libera R, bien emitiendo activamente una liberación o bien al ser «matado» al solicitar otro recurso. Si T2 es más joven, entonces se rearranca T2 pero con el mismo sello de tiempo que antes.

Así, en caso de conflicto, la transacción más antigua tiene prioridad. Dado que una transacción matada se revive con su sello de tiempo original, envejece y por tanto gana prioridad. Ningún sitio necesita conocer el estado de ubicación de todos los recursos. Todo lo que se necesita son los sellos de tiempo de las transacciones que solicitan sus recursos.

El método **herir-esperar** concede inmediatamente la solicitud de una transacción más antigua matando a la transacción más joven que está utilizando el recurso solicitado. Esto se muestra en la Figura 15.13b. En contraste con el método esperar-morir, una transacción nunca tiene que esperar por un recurso que está siendo usado por una transacción más joven.

Evitación del interbloqueo. Evitar los interbloqueos es una técnica en la cual se toma una decisión dinámicamente cuando una solicitud de asignación de recurso dada podría, si se concede, provocar un interbloqueo. [SING94b] hacen notar que evitar de forma distribuida un interbloqueo no es práctico por las siguientes razones:

1. Cada nodo debe seguirle la pista al estado global del sistema; esto requiere una cantidad sustancial de almacenamiento y sobrecarga de comunicaciones.
2. El proceso de comprobar que un estado global es seguro debe ser mutuamente exclusivo. De otro modo, dos nodos pueden estar considerando cada uno la solicitud de recurso de un proceso diferente y alcanzar concurrentemente la conclusión de que es seguro aceptar la solicitud, cuando de hecho si ambas solicitudes se conceden, sucederá un interbloqueo.
3. La comprobación de estados seguros conllevará una sobrecarga de procesamiento considerable para un sistema distribuido con un gran número de procesos y recursos.

Detección del interbloqueo. Con la detección del interbloqueo, a los procesos se les permite obtener recursos libres según lo deseen, y la existencia de un interbloqueo se determina después de que suceda. Si se detecta un interbloqueo, se selecciona a uno de los procesos que lo constituyen y se le solicita liberar los recursos necesarios para romper el interbloqueo.

```
if (e(T2) < e(T1))
    Parar_T2 ('esperar');
else
    Matar_T2 ('morir');
```

(a) Método esperar-morir

```
if (e(T2) < e(T1))
    Matar_T1 ('herir');
else
    Parar_T2 ('esperar');
```

(b) Método herir-esperar

Figura 15.13. Métodos de prevención del interbloqueo.

La dificultad de la detección de interbloqueo distribuida es que cada sitio conoce solamente sus recursos, mientras que el interbloqueo puede involucrar recursos distribuidos. Son posibles varios enfoques, dependiendo de si el sistema de control es centralizado, jerárquico o distribuido (Tabla 15.1).

Con el **control centralizado**, un sitio es el responsable de la detección de interbloqueos. Todos los mensajes de solicitud y liberación se envían al proceso central así como al proceso que controla el recurso particular. Dado que el proceso central tiene una imagen completa, está en posición de detectar un interbloqueo. Este enfoque requiere gran número de mensajes y es vulnerable al fallo del sitio central. Además, pueden detectarse los interbloqueos fantasma.

Con el **control jerárquico**, los sitios se organizan en una estructura en árbol, donde un sitio sirve como raíz del árbol. En cada nodo que no sea hoja, se recolecta información acerca de la ubicación de recursos de todos los nodos dependientes. Esto permite detectar el interbloqueo a niveles más bajos que el nodo raíz. Concretamente, un interbloqueo que involucre a un conjunto de recursos será detectado por el nodo que es ancestro común de todos los sitios cuyos recursos están entre los objetos en conflicto.

Con el **control distribuido**, todos los procesos cooperan en la tarea de detectar interbloqueos. En general, esto significa que debe intercambiarse una considerable cantidad de información, con sellos de tiempo; por tanto la sobrecarga es significativa. [RAYN88] cita varios enfoques basados en control distribuido y [DATT90] proporciona un examen detallado de un enfoque.

Veamos ahora un ejemplo de un algoritmo distribuido de detección de interbloqueo ([DATT92], [JOHN91]). El algoritmo trata con un sistema distribuido de base de datos en el cual cada sitio mantiene una porción de la base de datos y las transacciones pueden ser iniciadas desde cada sitio. Una transacción puede tener como máximo una solicitud de recurso pendiente. Si una transacción necesita

Tabla 15.1. Estrategias distribuidas de detección de interbloqueo.

Algoritmos centralizados		Algoritmos jerárquicos		Algoritmos distribuidos	
A favor	**En contra**	**A favor**	**En contra**	**A favor**	**En contra**
• Son conceptualmente sencillos y fáciles de implementar • Un sitio central tiene información completa y puede resolver interbloqueos óptimamente	• Sobrecarga considerable en comunicaciones; cada nodo debe enviar información de estado al nodo central • Vulnerable al fallo del nodo central	• No vulnerable a un único punto de fallo • La actividad de resolución de interbloqueos está limitada si los interbloqueos más probables están localizados	• Puede ser difícil configurar el sistema para que los interbloqueos más probables estén localizados; de otro modo puede realmente haber más sobrecarga que en la solución distribuida	• No vulnerable a un único punto de fallo • No se agobia a ningún nodo con la actividad de detección de interbloqueos	• La resolución de interbloqueos es incómoda porque varios sitios pueden detectar el mismo interbloqueo y pueden no considerar otros nodos involucrados en el interbloqueo • Los algoritmos son difíciles de diseñar debido a las consideraciones de tiempo

más de un objeto de datos, el segundo objeto de datos puede solicitarse solamente después de que el primero se haya concedido.

Asociados con cada objeto de datos *i* en un sitio hay dos parámetros: un identificador único D_i, y una variable Bloqueado_por(D_i). Esta última variable tiene el valor nulo si el objeto de datos no está bloqueado por ninguna transacción; en caso contrario su valor es el identificador de la transacción que lo bloquea.

Asociados con cada transacción *j* en un sitio hay cuatro parámetros:

- Un identificador único T_j.
- La variable Poseído_por(T_j), que se pone a nulo si la transacción T_j está ejecutando o en un estado Lista. En caso contrario, su valor es el identificador de la transacción que tiene el objeto de datos solicitado por la transacción T_j.
- La variable Espera_por(T_j), que tienen valor nulo si la transacción T_j no está esperando por ninguna otra transacción. En caso contrario, su valor es el identificador de la transacción que está en cabeza de la lista ordenada de transacciones que están bloqueadas.
- Una cola Solicitud_Q(T_j), que contiene todas las solicitudes pendientes por objetos de datos en posesión de T_j. Cada elemento en la cola tiene la forma (T_k, D_k), donde T_k es la transacción solicitante y D_k es el objeto de datos en posesión de T_j.

Por ejemplo, suponga que la transacción T_2 está esperando por el objeto de datos en posesión de T_1, la cual, a su vez, espera por un objeto de datos en posesión de T_0. Entonces los parámetros relevantes tienen los siguientes valores:

Transacción	Espera_por	Poseído_por	Solicitud_Q
T_0	nulo	nulo	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	nulo

Este ejemplo resalta la diferencia entre Espera_por(T_i) y Poseído_por(T_i). Ningún proceso puede continuar hasta que T_0 libere el objeto de datos que precisa T_1, que podrá entonces ejecutar y liberar el objeto de datos que precisa T_2.

La Figura 15.14 muestra el algoritmo utilizado para la detección de interbloqueo. Cuando una transacción realiza una solicitud de bloqueo de un objeto de datos, con un proceso servidor asociado con ese objeto de datos se concede o bien deniega la solicitud. Si la solicitud no se concede, el proceso servidor devuelve la identidad de la transacción que tiene el objeto de datos.

Cuando la transacción solicitante recibe una respuesta de concesión, bloquea el objeto de datos. En caso contrario, la transacción solicitante actualiza su variable Poseído_por con la identidad de la transacción que tiene el objeto de datos. Añade su identidad a la cola Solicitud_Q de la transacción poseedora. Actualiza su variable Espera_por a la identidad de la transacción poseedora (si dicha transacción no está esperando) o bien a la identidad de la variable Espera_por de la transacción que la posee. De este modo, la variable Espera_por toma el valor de la transacción que en última instancia está bloqueando la ejecución. Finalmente, la transacción solicitante emite un mensaje de actualización a todas las transacciones en su propia cola Solicitud_Q para modificar todas las variables Espera_por afectadas por este cambio.

```
/* Objeto de datos Dj recibiendo una mirar_solicitud(Ti) */
    if (Bloqueado_por(Dj) == null)
        enviar(concedido);
    else
    {
        enviar no concedido a Ti;
        enviar Bloqueado_por (Dj) a Ti
    }
/* La transacción Ti hace una solicitud mirar por el objeto Dj */
    enviar mirar_solicitud (Ti) a Dj;
    esperar a concedido / no concedido;
    if (concedido)
    {
        Bloqueado_por (Dj) = Ti;
        Poseído_por(Ti) = _;

    }
    else /* Suponer que Dj está en uso por la transacción Tj */
    {
        Poseído_por (Ti) = Tj;
        Encolar(Ti, Solicitud_Q(Tj));
        if (Esperar_por(Tj) == null)
            Esperar_por (Ti) = Tj;
        else
            Esperar_por (Ti) = Esperar_por (Tj);
        Actualizar(Esperar_por (Ti), Solicitud_Q (Ti));
    }

/* Transacción Tj recibiendo un mensaje de actualización */
    if (Esperar_por (Tj) != Esperar_por (Ti))
        Esperar_por (Tj) = Esperar_por (Ti);
    if (intersección(Esperar_por (Tj), Solicitud_Q (Tj)) = null)
        Actualizar(Esperar_por (Ti), Solicitud_Q (Tj);
    else
    {
        DECLARAR INTERBLOQUEO;
        /* Iniciar resolución de interbloqueo como sigue */
        /* Tj se escoge como la transacción a abortar */
        /* Tj liberar todos los objetos de datos que posee */
        enviar Limpiar (Tj, Poseído_por (Tj));
        asignar cada objeto de datos Di poseido por Tj al primer solicitante
        Tk en Solicitud_Q (Tj);
        for (cada transacción Tn en Solicitud_Q (Tj) solicitando un objeto de
        datos Di poseído por Tj)
        {
            Encolar(Tn, Solicitud_Q (Tk));
        }
    }
/* Transacción Tk recibiendo un mensaje Limpiar(Tj,Tk) */
eliminar la tupla que tenga Tj como transacción solicitante de Solicitud_Q (Tk);
```

Figura 15.14. Un algoritmo distribuido de detección de interbloqueo.

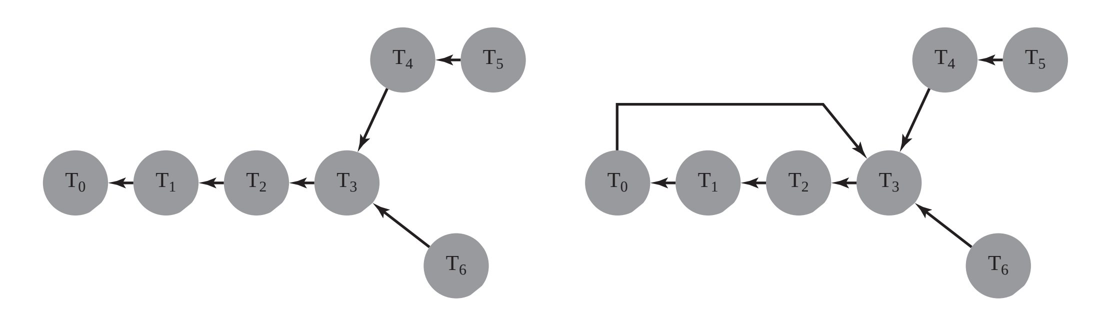
Cuando una transacción recibe un mensaje de actualización, actualiza su variable Espera_por para reflejar el hecho de que la transacción, sobre la cual está en última instancia esperando, está ahora bloqueada por una transacción más. Luego realiza el trabajo real de detección de interbloqueo comprobando si ahora está esperando por uno de los procesos que está a su vez esperando por él. Si no es así, se reenvía el mensaje de actualización. Si lo es, la transacción envía un mensaje de Limpiar a la transacción que posee el objeto de datos solicitado y entrega cada objeto de datos que posee al primer solicitante en su cola Solicitud_Q y encola las solicitudes restantes en la nueva transacción.

En la Figura 15.15 se muestra un ejemplo de la operación del algoritmo. Cuando T_0 realiza una solicitud por un objeto de datos poseído por T_3, se crea un ciclo. T_0 emite un mensaje de actualización que se propaga de T_1 a T_2 a T_3. En este punto T_3, descubre que la intersección de sus variables Espera_por y Solicitud_Q no es vacía. T_3 envía un mensaje de limpiar un a T_2 de manera que T_3 se elimina de Solicitud_Q(T_2), y libere al objeto de datos que posee, activando T_4 y T_6.

INTERBLOQUEO EN COMUNICACIÓN DE MENSAJES

Espera mutua. El interbloqueo en comunicación de mensajes sucede cuando cada proceso de un grupo está esperando por un mensaje de otro miembro del grupo y no hay mensajes en tránsito.

Para analizar esta situación en más detalle, definimos el conjunto de dependencia (DS) de un proceso. Para un proceso P_i que está parado, esperando por un mensaje, DS(P_i) consiste en todos los procesos desde los cuales P_i está esperando un mensaje. Típicamente, P_i puede continuar si cualquiera de los mensajes esperados llega. Una formulación alternativa es que P_i solamente puede continuar después de que lleguen todos los mensajes esperados. La primera situación es la más común y la que aquí se considera.



Transación	Esperar_por	Poseído_por	Solicitud_Q
T_0	nulo	nulo	nulo
T_1	T_0	T_0	T_2
T_2	T_0	T_1	T_3
T_3	T_0	T_2	T_4, T_6
T_4	T_0	T_3	T_5
T_5	T_0	T_4	nulo
T_6	T_0	T_3	nulo

(a)

Transación	Esperar_por	Poseído_por	Solicitud_Q
T_0	T_0	T_3	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	T_3
T_3	T_0	T_2	T_4, T_6, T_0
T_4	T_0	T_3	T_5
T_5	T_0	T_4	nulo
T_6	T_0	T_3	nulo

(b)

Figura 15.15. Ejemplo de algoritmo distribuido de detección de interbloqueo de la Figura 15.14.

Con la definición precedente, un interbloqueo en un conjunto S de procesos puede definirse como sigue:

1. Todos los procesos en S están parados, esperando mensajes.
2. S contiene el conjunto de dependencias de todos los procesos en S.
3. No hay mensajes en tránsito entre los miembros de S.

Cualquier proceso en S está interbloqueado porque nunca podrá recibir un mensaje que los desbloquee.

En términos gráficos, hay una diferencia entre interbloqueo de mensajes e interbloqueo de recursos. Con el interbloqueo de recursos, existe interbloqueo si hay un bucle cerrado, o ciclo, en el gráfico que describe las dependencias de los procesos. En el caso de recursos, un proceso depende de otro si el segundo posee un recurso que el primero necesita. Con el interbloqueo de mensajes, la condición para el interbloqueo es que todos los sucesores de un miembro de S estén asimismo en S.

La Figura 15.16 ilustra este punto. En la Figura 15.16a, P_1 está esperando por un mensaje de P_2 o de P_5; P_5 no está esperando por ningún mensaje y por tanto puede enviar un mensaje a P_1, que se libera. Como resultado, los enlaces (P_1, P_5) y (P_1, P_2) se borran. La Figura 15.16b añade una dependencia: P_5 está esperando por un mensaje de P_2, que está esperando por un mensaje de P_3, que está esperando por un mensaje de P_1, que está esperando por un mensaje de P_2. Así, existe interbloqueo.

Como con el interbloqueo de recursos, el interbloqueo de mensajes puede ser atacado por prevención o detección. [RAYN88] da algunos ejemplos.

No disponibilidad de *buffers* de mensaje. Otra forma en la que puede ocurrir interbloqueo en un sistema de paso de mensajes tiene que ver con la ubicación de *buffers* para el almacenamiento de mensajes en tránsito. Este tipo de interbloqueo es bien conocido en redes de datos de encaminamiento de paquetes. Primero examinamos este problema en el contexto de una red de datos y luego lo veremos desde el punto de vista de un sistema operativo distribuido.

La forma más simple de interbloqueo en una red de datos es el interbloqueo almacenar-y-reenviar directo y puede ocurrir si un nodo de encaminamiento de paquetes usa un depósito común para todos los *buffers* del cual se asignan *buffers* a paquetes por demanda. La Figura 15.17a muestra una situación en la cual todos los *buffers* del nodo A están ocupados con paquetes destinados a B. Lo opuesto es cierto en B. Ningún nodo puede aceptar más paquetes porque todos sus *buffers* están ocupados. Por tanto ningún nodo puede transmitir o recibir en ningún enlace.

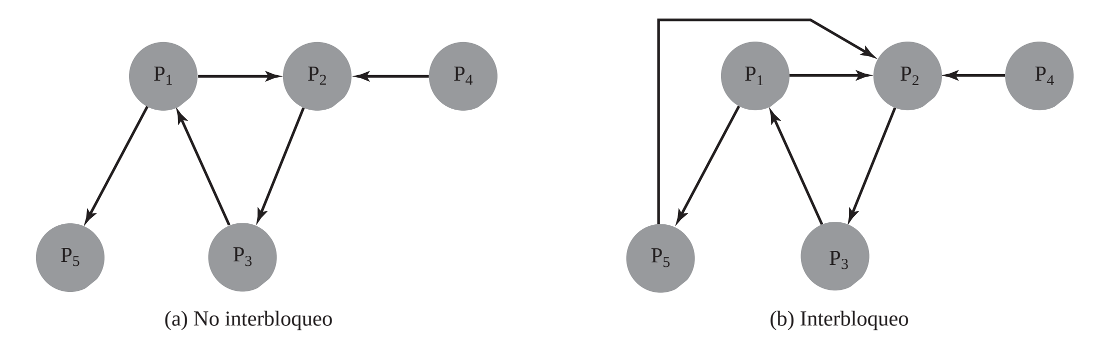
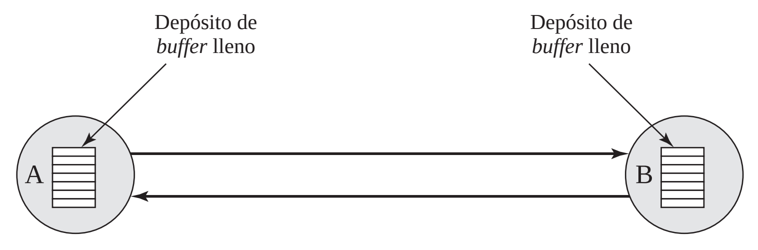
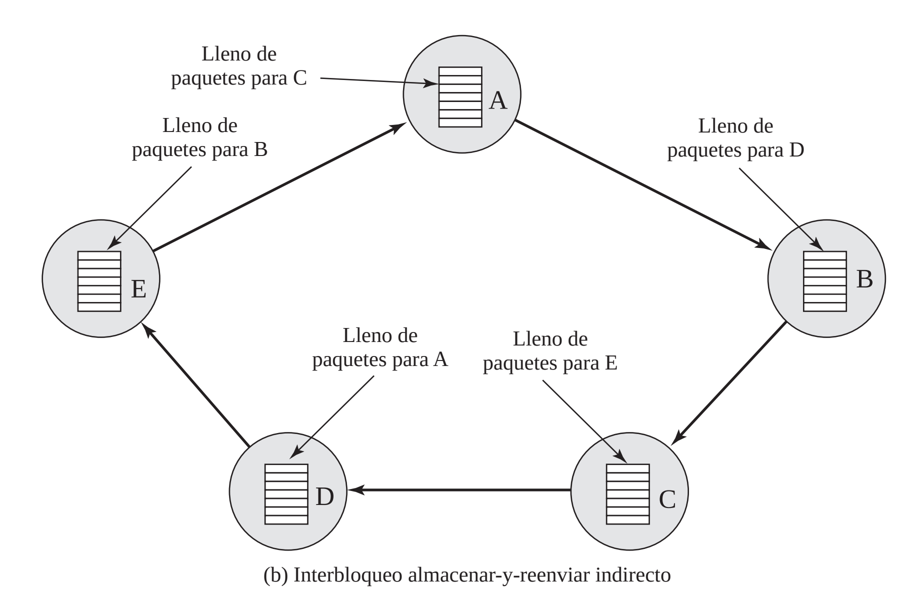


Figura 15.16. Interbloqueo en comunicación de mensajes.



(a) Interbloqueo almacenar-y-reenviar directo



(b) Interbloqueo almacenar-y-reenviar indirecto

Figura 15.17. Interbloqueo almacenar-y-reenviar.

El interbloqueo almacenar-y-reenviar directo puede preverse no permitiendo que todos los *buffers* terminen dedicados a un único enlace. Usando *buffers* de tamaño fijo separados, uno por enlace, se conseguirá la prevención. Incluso si se utiliza un depósito común de *buffers*, se evita el interbloqueo si se impide que un único enlace pueda ocupar todo el espacio de *buffers*.

Una forma más sutil de interbloqueo, interbloqueo almacenar-y-reenviar indirecto, se ilustra en la Figura 15.17b. Para cada nodo, la cola al nodo adyacente en una dirección está repleta con paquetes destinados al siguiente nodo más allá. Una manera sencilla de prevenir este tipo de interbloqueo es emplear un depósito estructurado de *buffers* (Figura 15.18). Los *buffers* se organizan de manera jerárquica. El depósito de memoria a nivel 0 no está restringido; cualquier paquete entrante puede almacenarse en él. Del nivel 1 al nivel N (donde N es el máximo número de saltos en cualquier camino de la red), los *buffers* se reservan del siguiente modo: los *buffers* del nivel *k* se reservan para paquetes que ya han viajado como mínimo *k* saltos. Así, en condiciones de alta carga, los *buffers* se llenan progresivamente desde el nivel 0 al nivel N. Si todos los *buffers* hasta el nivel *k* están llenos, los paquetes entrantes que han cubierto *k* o menos saltos se pierden. Puede mostrarse [GOPA85] que esta estrategia elimina los interbloqueos almacenar-y-reenviar tanto directos como indirectos.

El problema de interbloqueo que se acaba de describir sería tratado dentro del contexto de la arquitectura de comunicaciones, típicamente en el nivel de red. El mismo tipo de problema puede suce-

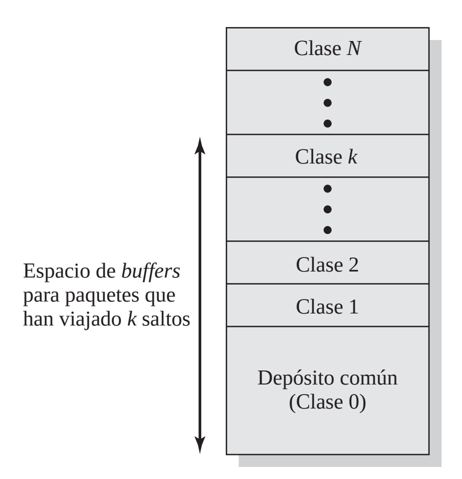


Figura 15.18. Depósito de *buffers* estructurado para la prevención de interbloqueo.

der en un sistema operativo distribuido que utiliza paso de mensajes para la comunicación entre procesos. Concretamente, si la operación de envío no es bloqueante, se necesita un *buffer* para contener los mensajes salientes. Podemos pensar que el *buffer* utilizado para contener los mensajes enviados desde el proceso X al proceso Y es el canal de comunicaciones entre X e Y. Si este canal tiene una capacidad finita (tamaño de *buffer* finito), es posible que la operación de envío termine con la suspensión del proceso. Esto es, si el *buffer* es de tamaño *n* y actualmente hay *n* mensajes en tránsito (no recibidos todavía por el proceso destinatario), la ejecución de un envío más bloqueará al proceso remitente hasta que una recepción abra espacio en el *buffer*.

La Figura 15.19 ilustra cómo el uso de canales finitos puede llevar al interbloqueo. La figura muestra dos canales, cada uno con capacidad para cuatro mensajes, uno del proceso X al proceso Y y uno del Y al X. Si hay exactamente cuatro mensajes en tránsito en cada uno de los canales y ambos X e Y intentan una transmisión más antes de ejecutar una recepción, ambos se suspenden y aparece el interbloqueo.

Si es posible establecer límites superiores al número de mensajes estarán en tránsito entre cada pareja de procesos en el sistema, entonces la estrategia de prevención obvia sería ubicar tantos *buffers* como se necesiten para todos estos canales. Esto puede ser un derroche extremadamente grande

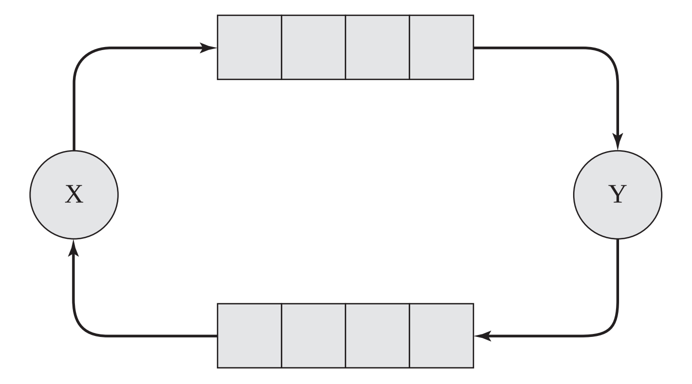


Figura 15.19. Interbloqueo de comunicaciones en un sistema distribuido.

y por supuesto requiere este conocimiento anticipado. Si las necesidades no pueden conocerse con anticipación, o si la ubicación basada en límites superiores es definitivamente prohibitiva, se necesita alguna técnica de estimación para optimizar la ubicación. Puede demostrarse que este problema no tiene solución en el caso general; en [BARB90] se sugieren algunas estrategias heurísticas para tratar esta situación.

15.5. RESUMEN

Un sistema operativo distribuido puede proporcionar migración de procesos. Esta es la transferencia de suficiente cantidad del estado de un proceso de una máquina a otra para que el proceso pueda ejecutar en la máquina destino. La migración de procesos puede utilizarse para el equilibrado de carga, para mejorar las prestaciones minimizando la actividad de las comunicaciones, para mejorar la disponibilidad o para permitir que los procesos accedan a facilidades remotas especiales.

Con un sistema distribuido, es a menudo importante establecer información de estado global, para resolver competencia por recursos y coordinar procesos. Dado lo variable e impredecible del retardo en la transmisión de mensajes, debe tenerse cuidado en asegurar que diferentes procesos se ponen de acuerdo en el orden en el que suceden los eventos.

La gestión de procesos en un sistema distribuido incluye a los servicios para proporcionar exclusión mutua y para tomar acciones en caso de interbloqueo. En ambos casos, los problemas son más complejos que aquellos en un sistema único.

15.6. LECTURA RECOMENDADA

[GALL00] y [TEL01] cubren todos los temas de este capítulo.

[MILO00] es una amplia y detallada recopilación sobre mecanismos de migración de procesos e implementaciones. Otras recopilaciones útiles son [ESKI90] y [SMIT88]. [NUTT94] describe varias implementaciones de migración de procesos en el SO. [FIDG96] recopila varias soluciones a la ordenación de eventos en sistemas distribuidos y concluye que el preferido es el enfoque general descrito en este capítulo.

Algoritmos para la gestión distribuida de procesos (exclusión mutua, interbloqueo) pueden encontrarse en [SINH97] y [RAYN88]. [RAYN90], [GARG02] y [LYNC96] contienen un tratamiento más formal.

ESKI90 Eskicioglu, M.«Design Issues of Process Migration Facilities in Distributed Systems.» *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, Verano 1990.

FIDG96 Fidge, C. «Fundamentals of Distributed System Observation.» IEEE Software, Noviembre 1996.

GALL00 Galli, D. *Distributed Operating Systems: Concepts and Practice*. Upper Saddle River, NJ: Prentice Hall, 2000.

GARG02 Garg, V. *Elements of Distributed Computing.* New York:Wiley, 2002.

LYNC96 Lynch, N. *Distributed Algorithms*. San Francisco, CA: Morgan Kaufmann, 1996.

MILO00 Milojicic, D.; Douglis, F.; Paindaveine, Y.; Wheeler, R. y Zhou, S. «Process Migration.» ACM *Computing Surveys*, Septiembre 2000.

NUTT94 Nuttal, M. «A Brief Survey of Systems Providing Process or Object Migration Facilities.» *Operating Systems Review*, Octubre 1994.

RAYN88 Raynal, M. *Distributed Algorithms and Protocols*. New York:Wiley, 1988.

RAYN90 Raynal, M. y Helary, J. *Synchronization and Control of Distributed Systems and Programs*. New York:Wiley, 1990.

SING94 Singhal, M. y Shivaratri, N. *Advanced Concepts in Operating Systems*. New York: McGraw-Hill, 1994.

SINH97 Sinha, P. *Distributed Operating Systems*. Piscataway, NJ: IEEE Press, 1997.

SMIT88 Smith, J. «A Survey of Process Migration Mechanisms.» *Operating Systems Review*, Julio 1988.

TEL01 Tel, G. *Introduction to Distributed Algorithms*. Cambridge: Cambridge University Press, 2001.

15.7. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

canal	exclusión mutua distribuida	migración de procesos
desalojo	instantánea	transferencia expulsiva
estado global	interbloqueo distribuido	transferencia no expulsiva

CUESTIONES DE REPASO

15.1. Enumere algunas de las razones para implementar migración de procesos.

15.2. ¿Cómo se manipula el espacio de direcciones del proceso durante la migración del proceso?

15.3. ¿Cuáles son las motivaciones para la migración de proceso expulsiva y no expulsiva?

15.4. ¿Por qué es imposible determinar un estado global real?

15.5. ¿Cuál es la diferencia entre exclusión mutua distribuida basada en un algoritmo centralizado y basada en un algoritmo distribuido?

15.6. Defina los dos tipos de interbloqueo distribuido

PROBLEMAS

15.1. La política de volcado se describe en la subsección sobre estrategias de migración de procesos de la Sección 15.1.

a) Desde la perspectiva del origen, ¿a qué otra estrategia se asemeja el volcado?

b) Desde la perspectiva del destino, ¿a qué otra estrategia se asemeja el volcado?

15.2. En la Figura 15.9, se indica que los cuatro procesos asignan una ordenación de $\{a, q\}$ a los dos mensajes, incluso aunque q llega antes que a a P_3. Explore el algoritmo para demostrar la verdad de esta afirmación.

15.3. Para el algoritmo de Lamport, ¿hay alguna circunstancia bajo la cual P_i puede ahorrarse la transmisión del mensaje Respuesta?

15.4. Para el algoritmo de exclusión mutua de [RICA81],

a) Pruebe que se cumple la exclusión mutua.

b) Si los mensajes no llegan en el orden en que fueron enviados, el algoritmo no garantiza que las secciones críticas sean ejecutadas en el orden de sus solicitudes. ¿Es posible la inanición?

15.5. En el algoritmo de exclusión mutua de paso de testigo, ¿se utiliza el sello de tiempo para restablecer los relojes y corregir las derivas, como en los algoritmos de cola distribuida? Si no es así, ¿cuál es la función del sello de tiempo?

15.6. Para el algoritmo de exclusión mutua de paso de testigo, pruebe que

a) garantiza la exclusión mutua

b) evita el interbloqueo

c) es equitativo

15.7. En la Figura 15.11b, explique por qué la segunda línea no puede ser simplemente «solicitud(j) = t».

CAPÍTULO 16

Seguridad

◆ ◆

El área de la seguridad de sistemas informáticos es muy amplia y cubre tanto controles físicos y administrativos como controles lógicos. En este capítulo, nos centraremos únicamente en las consideraciones relativas a las herramientas de seguridad remota y controles lógicos. La Figura 16.1 sugiere el ámbito de responsabilidad de estas herramientas. Comenzaremos examinando el tipo de peligros a los cuales se enfrenta una instalación informática y de comunicaciones. A continuación, el grueso del capítulo tratará de las herramientas específicas que se pueden utilizar para mejorar la seguridad. La Sección 16.2 cubre las estrategias tradicionales para garantizar la seguridad informática, que se basan en la protección de determinados recursos, incluyendo memoria y datos. Seguidamente, veremos los peligros que suponen los individuos que intentan superar estos mecanismos de protección. La siguiente sección examina los peligros derivados de virus y mecanismos similares. A continuación, se examinará el concepto de sistemas confiables. Finalmente, un apéndice de este capítulo introduce al lector al cifrado, que es la herramienta básica usada para muchas aplicaciones de seguridad.

◆ ◆

16.1. AMENAZAS DE SEGURIDAD

Para comprender los diferentes peligros existentes a nivel de seguridad, es necesario comenzar por la definición de los requisitos de seguridad. La seguridad de los sistemas informáticos y de la red va dirigida a cuatro requisitos básicos:

- **Confidencialidad.** Requiere que la información de un sistema informático sólo se encuentre accesible para lectura para aquellas partes que estén autorizadas a este tipo de acceso. Este tipo de acceso incluye impresión, mostrado de datos y otras formas de observación, incluyendo la simple revelación de la existencia de un elemento.
- **Integridad.** Requiere que los contenidos de un sistema informático sólo podrán modificarse por las partes que se encuentran autorizadas. Las modificaciones incluyen escritura, cambio, modificación del estado, borrado y creación.
- **Disponibilidad.** Requiere que los componentes de un sistema informático estén disponibles para todas aquellas partes autorizadas.
- **Autenticación.** Requiere que el sistema informático sea capaz de verificar la identidad de los usuarios.

TIPOS DE PELIGROS

Los tipos de ataques contra la seguridad del sistema o de la red se clasifican mejor considerando las funciones de un sistema informático como si se tratase de un proveedor de información. En general, existe un flujo de información desde una fuente, pudiéndose tratar de un fichero o una región de memoria, a un destino, que puede ser otro fichero o puede ser el mismo usuario. Ese flujo virtual se muestra en la Figura 16.2a. El resto de elementos de la figura muestran las siguientes cuatro categorías generales de ataques:

- **Interrupción.** Se destruye un componente del sistema o se encuentra no disponible o utilizable. Es un ataque centrado en la **disponibilidad**. Ejemplos de este tipo incluyen la destrucción de una pieza del hardware, como un disco duro, la interrupción del canal de comunicación o la eliminación del sistema gestor ficheros.
- **Intercepción.** Una parte no autorizada consiga acceso a un componente. Esto es un ataque dirigido hacia la **confidencialidad**. La parte no autorizada puede ser una persona, un programa o

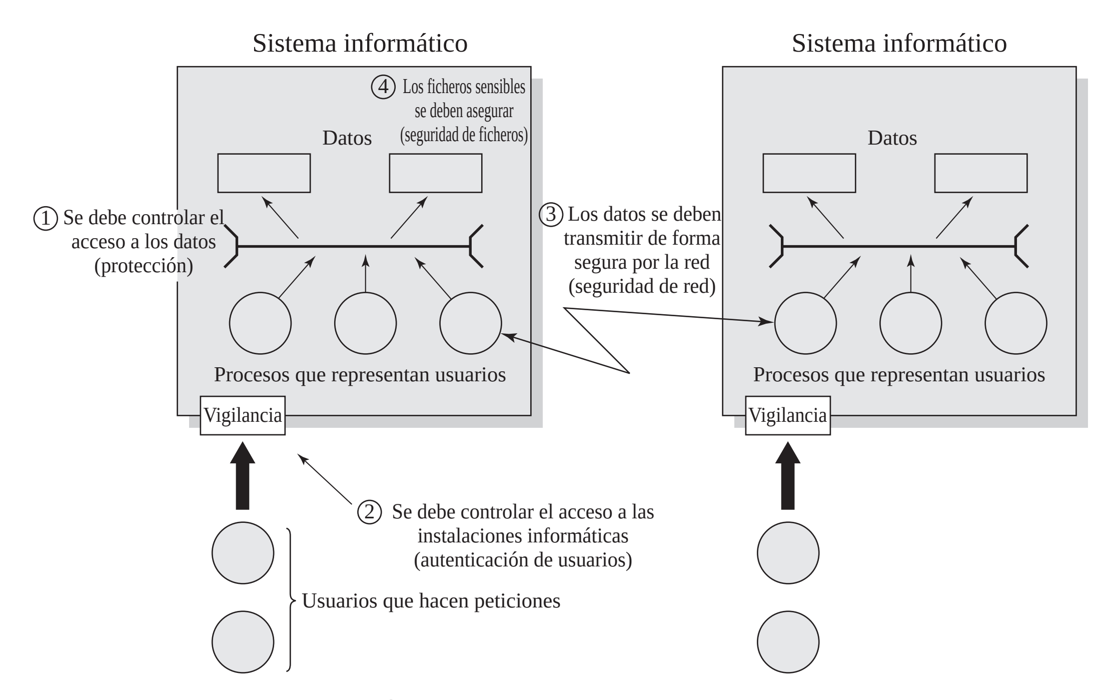


Figura 16.1. Ámbito de la seguridad informática [MAEK87].

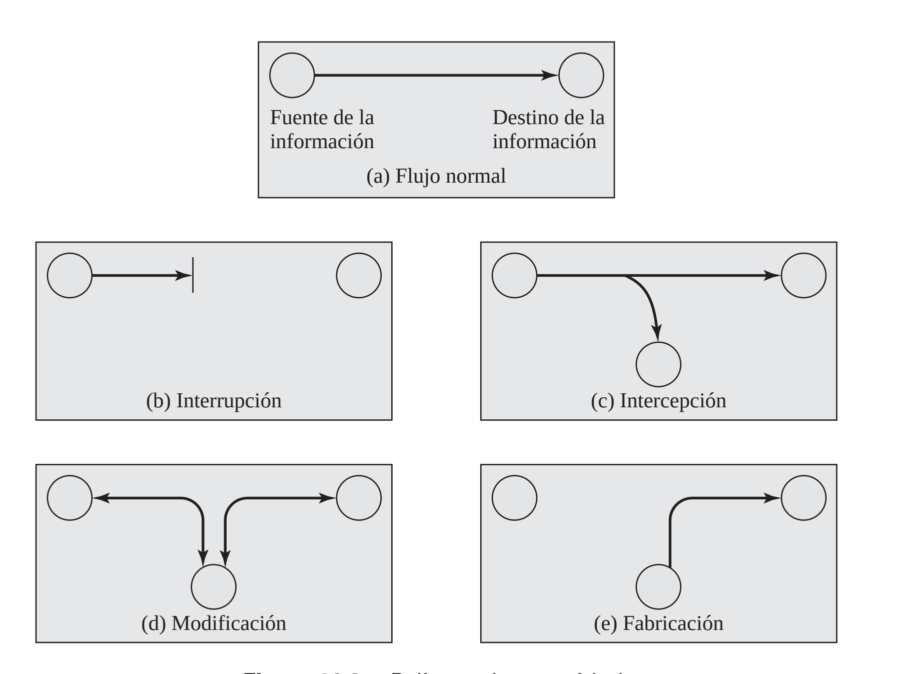


Figura 16.2. Peligros de seguridad.

un ordenador. Ejemplos de este estilo son la escucha en un canal de comunicación para capturar datos y la copia ilícita de ficheros o programas.

- **Modificación.** Un elemento no autorizado no sólo tiene acceso a un componente sino que también es capaz de modificarlo. Éste es un ataque que va dirigido hacia la **integridad**. Los ejemplos incluyen cambiar valores de un fichero de datos, alterar un programa para que exhiba un comportamiento diferente, y modificar el contenido de los mensajes que se transmiten por la red.
- **Fabricación.** Un elemento no autorizado inserta objetos extraños en el sistema. Estos son ataques contra la **autenticación**. Ejemplos de este tipo son la inserción de mensajes externos en una red o la inclusión de un registró en un fichero

COMPONENTES DE UN SISTEMA INFORMÁTICO

Los componentes de un sistema informático se pueden clasificar en hardware, software, datos y líneas de comunicaciones y red. En la Tabla 16.1 se indica la naturaleza de los peligros que afectan a cada una de estas categorías. A continuación revisaremos cada una por separado.

Hardware. El principal peligro del hardware de un sistema informático se encuentra en el área de la disponibilidad. El hardware es el componente más vulnerable a ataques y también es el menos accesible a una manipulación remota. Los principales peligros incluyen daño accidental o deliberado a los equipos, así como el robo. La proliferación de ordenadores personales y estaciones de trabajo y el incremento en el uso de redes de área local incrementan las potenciales pérdidas en esta área. Para hacer a frente estos peligros se necesitan medidas de seguridad física y administrativa.

Software. Lo que hace del hardware del sistema informático algo útil para negocios e individuos son el sistema operativo, las utilidades y los programas de aplicación. Existen diferentes tipos de peligros a tener en cuenta.

Tabla 16.1. Peligros de seguridad y componentes.

	Disponibilidad	Privacidad	Integridad/Autenticación
Hardware	Equipamiento robado o deshabitado, por lo tanto denegación de servicio.		
Software	Borrado de programas, denegación de acceso a los usuarios.	Copia no autorizada de software.	Modificación de un programa, bien para hacer que falle durante la ejecución o para que realice una tarea diferente.
Datos	Borrar ficheros, denegación de acceso a los usuarios.	Lectura no autorizada de datos. Un análisis estadístico de los datos que revele la información subyacente.	Modificación de los ficheros existentes o creación de nuevos ficheros.
Líneas de comunicación	Borrado o destrucción de mensajes. Las líneas de comunicación o redes no se encuentran disponibles.	Lectura de mensajes. Observación de los patrones de tráfico de mensajes.	Modificación, borrado, reordenación o duplicación de mensajes. Fabricación de mensajes falsos.

Un peligro importante en relación al software es el referente a la disponibilidad. El software, especialmente el software de aplicación, es a menudo fácil de borrar. De la misma forma, también puede verse alterado o dañado hasta dejarlo inservible. Para mantener una alta disponibilidad, resulta necesaria una gestión de la configuración del software cuidadosa, que incluya realización de copias de respaldo (*backups*) y actualización a las versiones más recientes del software. Una cuestión más complicada es la referente a la modificación del software que hace que el programa sigua funcionando pero que su comportamiento sea diferente al que realizaba anteriormente, lo cual implica un peligro de integridad y autenticación. Los virus informáticos y los ataques similares se encuentran dentro de esta categoría y se tratarán más adelante en este capítulo. El último problema es el relativo a la privacidad. A pesar de que existen diferentes medidas disponibles, el problema de la copia no autorizada de software aún no se encuentra resuelto.

Datos. La seguridad relativa al hardware y al software se encuentra habitualmente dentro de los cometidos de los profesionales de administración del sistema informático, o en el caso de pequeñas instalaciones, en los propietarios de ordenadores personales. Un problema mucho más amplio es el relativo a la seguridad de los datos, o que incluye los ficheros y cualquier otro tipo de datos controlados por los individuos, grupos, u organizaciones.

Los aspectos de seguridad relativos a los datos son muy amplios, incluyendo la disponibilidad, la privacidad y la integridad. El caso de la disponibilidad, se centra en la destrucción de ficheros de datos, que puede ocurrir de forma accidental o maliciosa.

Un aspecto evidente en lo concerniente a la privacidad es, por supuesto, la lectura no autorizada de ficheros de datos o bases de datos, y en esta área se han volcado mayor cantidad de esfuerzos e investigación que en cualquier otro área de la seguridad informática. Un peligro menos obvio para la privacidad incluye el análisis de datos, y se manifiesta en el uso de, las así llamadas, bases de datos estadísticas, que contienen información resumida o agregada. Presumiblemente, la existencia de información agregada no es un peligro para la privacidad de los individuos. Sin embargo, a medida que crece el uso de las bases de datos estadísticas, existe un incremento potencial de la filtración de información personal. En esencia, las características de individuos concretos se pueden identificar a través de un análisis minucioso. Por poner un ejemplo sencillo, si un registro de una tabla agregada incluye los ingresos correspondientes a A, B, C, y D y otro registro suma los ingresos de A, B, C, D, y E, la diferencia entre los dos valores sería los ingresos de E. Este problema se acrecienta con el creciente deseo de combinar conjuntos de datos. En muchos casos, el cruce de diversos conjuntos de información hasta los niveles apropiados de relevancia para un problema determinado, implican descender hasta nivel de unidades elementales para poder construir los datos necesarios. De esta forma, las unidades elementales, que sí están sujetas a consideraciones de privacidad, se encuentran disponibles en diferentes pasos del proceso de estos conjuntos de datos.

Para finalizar, la integridad de datos es un aspecto clave en muchas instalaciones. La modificación de los ficheros de datos puede llevar a una serie de consecuencias que van desde problemas menores hasta desastrosos.

Líneas de comunicaciones y redes. Un mecanismo útil para la clasificación de los ataques de seguridad a la redes es en base a los términos de ataques pasivos y ataques activos. Un ataque pasivo intenta aprender o hacer uso de la información del sistema, pero no afecta a los recursos del mismo. Un ataque activo intenta alterar los recursos del sistema o afectar a su operativa.

Los **ataques pasivos** son el espionaje o la monitorización de las transmisiones. El objetivo del oponente es obtener información de qué se está transmitiendo. Los dos tipos de ataques pasivos son la lectura de los contenidos de los mensajes y el análisis de tráfico.

La **lectura de los contenidos de los mensajes** es fácil de comprender (Figura 16.3a). Una conversación telefónica, un mensaje de correo electrónico o la transferencia de un fichero pueden conte-

ner información sensible y confidencial. Sería deseable evitar que cualquier oponente tenga acceso a los contenidos de estas transmisiones.

Un segundo tipo de ataques pasivos, los ataques de **análisis de tráfico**, son más discretos (Figura 16.3b). Supongamos que tenemos un mecanismo para ocultar los contenidos de los mensajes u otro tráfico de información de forma que los oponentes, intrusos si capturan los mensajes, no puedan extraer la información que contiene. La técnica habitual para ocultar estos contenidos es el cifrado. Si queremos una protección basada en cifrado funcionando, un oponente aún podría observar cuáles son los patrones de estos mensajes. El oponente podría determinar la ubicación e identidad de los ordenadores que se comunican y podría observar la frecuencia y objeto de los mensajes que se intercambian. Esta información puede resultar útil para adivinar la naturaleza de las comunicaciones que se están llevando a cabo.

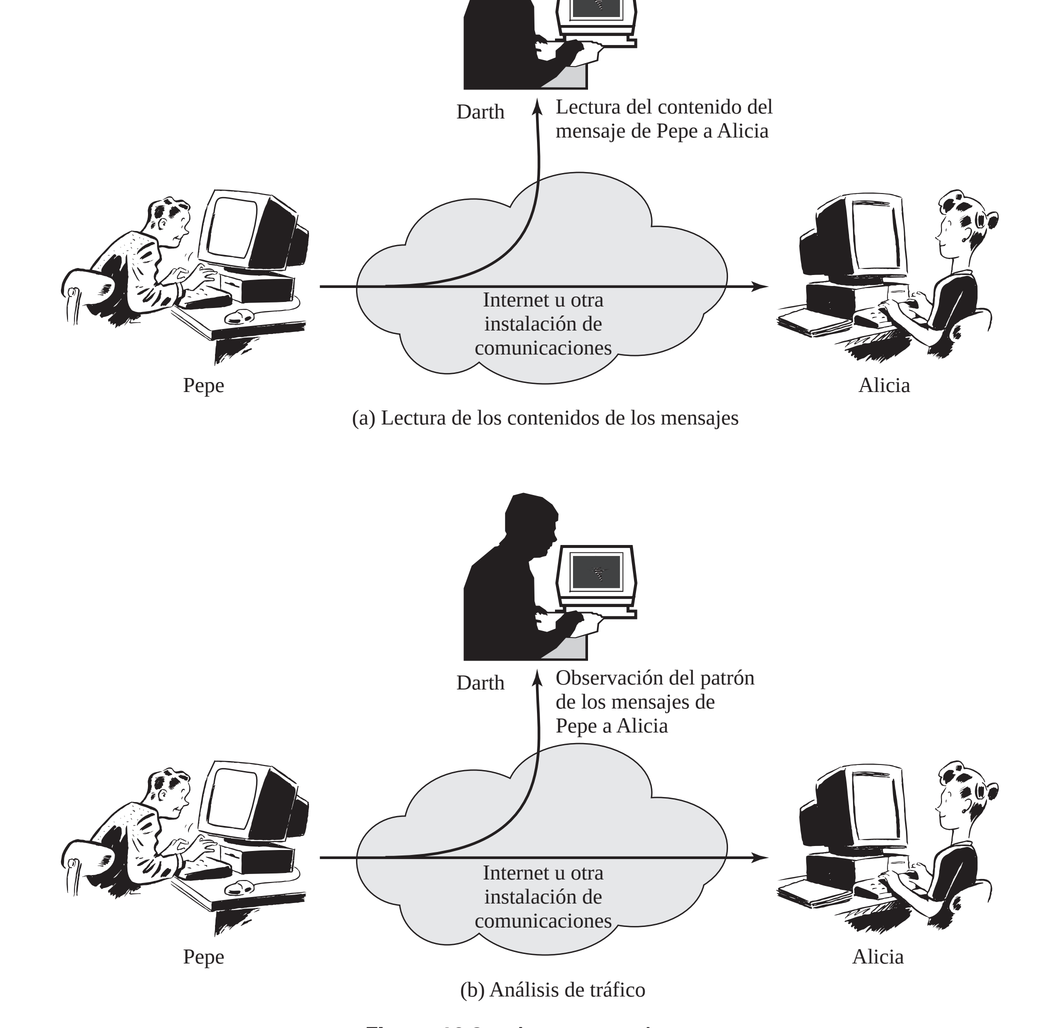


Figura 16.3. Ataques pasivos.

Los ataques pasivos son muy difíciles de detectar debido a que no implican ninguna alteración de los datos. Habitualmente, el tráfico de mensajes se envía y recibe de una manera aparentemente normal y ni el emisor ni el receptor se dan cuenta de que un tercer elemento ha leído los mensajes o analizado los patrones de tráfico. Sin embargo, es posible prevenir estos ataques, habitualmente por medio de cifrado. De esta forma, el énfasis que se hace sobre los ataques pasivos se centra en su prevención más que en su detección.

Los **ataques activos** implican algunas modificaciones en el flujo de datos o la creación de flujos de datos falsos y se pueden subdividir en cuatro diferentes categorías: enmascaramiento, reenvío, modificación de mensajes y denegación de servicio.

El **enmascaramiento** ocurre cuando el elemento intenta hacerse pasar por otro diferente (Figura 16.4a). Un ataque de enmascaramiento incluye habitualmente una de las otras formas de ataques activos. Por ejemplo, se pueden capturar las secuencias de autenticación y reenviarla posteriormente, después de que se haya intercambiado una secuencia válida, de forma que permita a un elemento con pocos privilegios obtener privilegios extra, suplantando a otro que los posea.

El **reenvío** implica la captura pasiva de una unidad de datos y su posterior retransmisión para producir un efecto no autorizado (Figura 16.4c).

La **modificación de mensajes** significa sencillamente que una parte de un mensaje válido se ha alterado, o que los mensajes se han borrado o reordenado, para producir un efecto no autorizado (Figura 16.4c). Por ejemplo, si se modifica un mensaje cuyo significado es «Permite a Pepe Pérez leer el fichero confidencial de cuentas» para que diga «Permite a Juanito Sánchez leer el fichero confidencial de cuentas».

La **denegación de servicio** previene o imposibilitar el uso normal o la gestión de las instalaciones de comunicaciones (Figura 16.4d). Este ataque puede tener un objetivo específico; por ejemplo, un elemento puede suprimir todos los mensajes dirigidos a un destino en particular (por ejemplo, el servicio de auditoría de seguridad). Otra forma de denegación de servicio es la desarticulación de toda la red, bien deshabilitándola o sobrecargándola con mensajes para degradar su rendimiento.

Los ataques activos presentan características opuestas a los ataques pasivos. Mientras que los ataques pasivos son difíciles de detectar, sí existen medidas que permiten evitar su éxito. Por otro lado, es bastante más difícil prevenir los ataques activos de forma completa, debido a que para poder hacerlo se requerirían protecciones físicas para todas las instalaciones de comunicaciones y todas las rutas. Sin embargo, el objetivo en este caso es la detección de dichos ataques y la recuperación de cualquier efecto o retraso que pudieran causar. Debido a que la detección tiene un efecto disuasorio, también esto contribuye a su prevención.

16.2. PROTECCIÓN

La introducción de la multiprogramación trajo consigo la posibilidad de compartir recursos entre los usuarios. Esta compartición implica no sólo procesador sino también:

- Memoria
- Dispositivos de E/S, como discos e impresoras
- Programas
- Datos

La posibilidad de compartir estos recursos introduce la necesidad de la protección. [PFLE97] apunta a que el sistema operativo debe ofrecer niveles de protección a lo largo del siguiente rango:

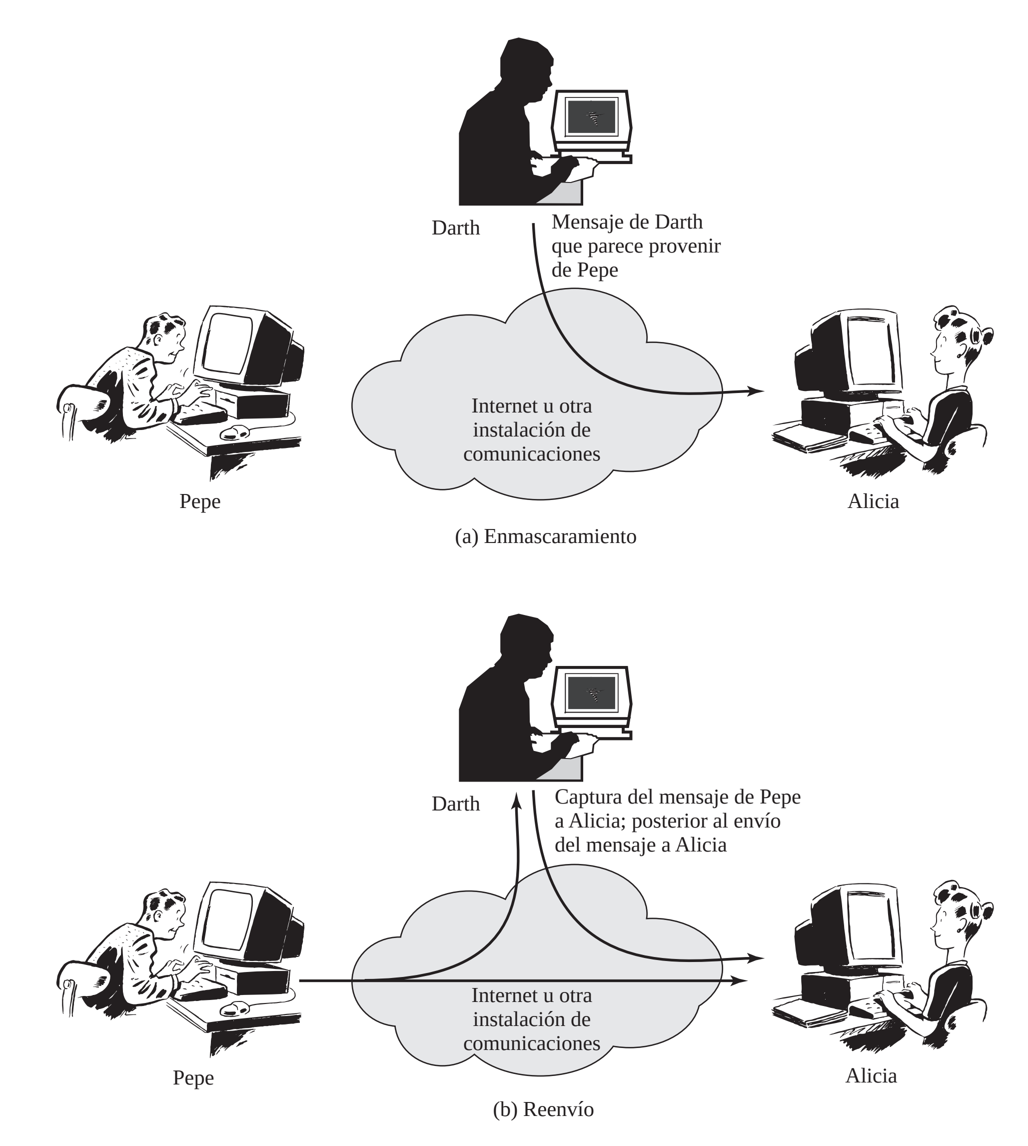


Figura 16.4. Ataques activos.

- **Sin protección alguna.** Apropiado por los procedimientos que son sensibles de ejecutar en instantes diferentes.
- **Aislamiento.** Esta estrategia implica que cada proceso opera de forma separada con otros procesos, sin compartición ni comunicación alguna. Cada proceso tiene su propio espacio de direcciones, ficheros, y otros objetos.
- **Compartición completa o sin compartición.** El propietario del objeto (por ejemplo, un fichero o un segmento de memoria) declara si va a ser público o privado. En el primer caso, cualquier proceso puede acceder al objeto; en el último, sólo los procesos del propietario pueden acceder a dicho objeto.
- **Compartición vía limitaciones acceso.** El sistema operativo verificar la permisibilidad de cada acceso por parte de cada usuario específico sobre cada objeto. El sistema operativo, por

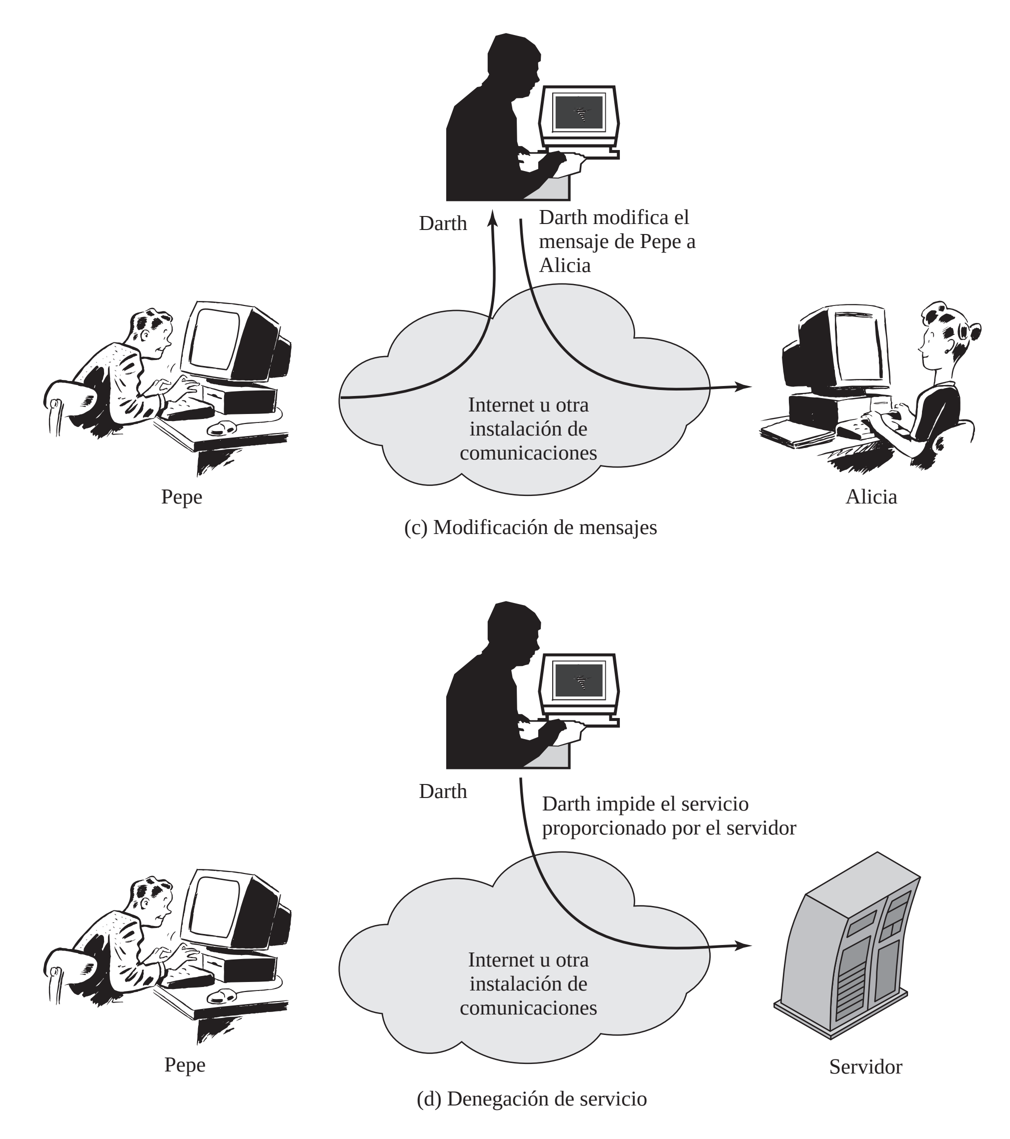


Figura 16.4. Continuación.

tanto, actúa como guardián o vigilante, entre los usuarios y los objetos, asegurando que sólo se producen los accesos autorizados.

- **Acceso vía capacidades dinámicas.** Permite la creación dinámica de derechos de acceso a los objetos.
- **Uso limitado de un objeto.** Esta forma de protección limita no sólo el acceso a un objeto sino también el uso que se puede realizar de dicho objeto. Por ejemplo, un usuario puede estar capacitado para ver un documento sensible, pero no para imprimirlo. Otro ejemplo puede ser que el usuario tenga acceso a una base datos para calcular estadísticas pero no para obtener determinados valores específicos.

Los puntos anteriores se han presentado en orden de dificultad aproximadamente creciente desde el punto de vista de su implementación, así como en el orden del detalle de protección que proporciona. Un sistema operativo determinado puede proporcionar diferentes grados de protección sobre distintos objetos, usuarios o aplicaciones.

El sistema operativo necesita equilibrar la necesidad de permitir compartición, que mejora la utilidad del sistema, con la necesidad de proteger los recursos de cada uno de los usuarios. En esta sección, vamos a mostrar algunos de los mecanismos mediante los cuales el sistema operativo puede proporcionar protección para estos objetos.

PROTECCIÓN DE LA MEMORIA

En un entorno multiprogramado, la protección de la memoria principal es esencial. Los aspectos importantes aquí no son sólo la seguridad, sino también el correcto funcionamiento de varios procesos que se pueden encontrar activos. Si un proceso puede escribir de forma inadvertida en el espacio memoria de otro proceso, este último no se ejecutará de forma correcta.

La separación de los espacios de memoria de los diferentes procesos es un requisito fundamental de los esquemas de memoria virtual. Ya sea en base a la segmentación o a la paginación o a la combinación de ambas, es necesario proporcionar unos mecanismos efectivos para gestionar la memoria principal. Si lo que se busca es el aislamiento completo, entonces el sistema operativo únicamente debe asegurar que a cada segmento o página se accede sólo por parte del proceso al que está asignada. Éste es un requisito fácil de alcanzar, asegurándose únicamente de que no hay entradas duplicadas en las tablas de páginas y/o segmentos.

Si se desea permitir la compartición, entonces el mismo segmento o página puede aparecer en más de una tabla. Ese tipo de compartición es más fácil de alcanzar en un sistema que soporte segmentación o la combinación de segmentación y paginación. En este caso, la estructura de segmentos es visible para la aplicación, y es la propia aplicación la que puede declarar que determinados segmentos se encuentren compartidos o no. En un entorno de paginación por tanto, resulta más difícil diferenciar entre estos dos tipos de memoria, debido a que la estructura de memoria es completamente transparente a la aplicación.

Un ejemplo del soporte hardware que se puede proporcionar para dar apoyo a la protección de memoria es el que proporcionan las máquinas zSeries de IBM, sobre las cuales se ejecuta el sistema operativo z/OS. Hay una clave de control de almacenamiento de 7-bits asociada a cada marco de página en memoria principal, que el sistema operativo puede fijar. Dos de estos bits indican si la página que está ocupando este marco se ha referenciado o cambiado; es el algoritmo de reemplazo de páginas el que los usa. El resto de bits se utilizan por parte de los mecanismos de protección: una clave de control de acceso de cuatro bits y un bit de protección de búsqueda. Las referencias a memoria del procesador o de la E/S vía DMA deben utilizar una clave que se ajuste para obtener privilegios de acceso a dicha página. El bit de protección de búsqueda indica si la clave de control de acceso se aplica a escrituras o a escrituras y lecturas indistintamente. En el procesador, hay una palabra de estado de programa (*program status word*, PSW), la cual contiene información de control relativa al proceso que está actualmente en ejecución. Incluida en esta palabra se encuentra la clave PSW de 4-bits. Cuando un proceso intenta acceder a una página o se arranca una operación de DNA sobre una página, la clave PSW actual se compara con el código de acceso. Una operación de escritura sólo se permite si los códigos coinciden. Si el bit de protección de búsqueda está puesto a 1, entonces la clave PSW debe coincidir con el código de acceso para las operaciones de lectura.

CONTROL DE ACCESO ORIENTADO A USUARIO

Las medidas de control de acceso para un sistema de procesamiento de datos se encuadran en dos diferentes categorías: aquellas asociadas al usuario y aquellas asociadas a los datos.

El control acceso por usuario se domina a veces, y de forma poco afortunada, como autenticación. Debido a que este término se utiliza de forma generalizada en la actualidad en el sentido de autenticación de mensajes, evitaremos su uso en este texto. De todas formas, se advierte al lector que esta utilización puede encontrarse a menudo en la literatura.

La técnica más común para el control de acceso por usuario en un sistema compartido o en un servidor es el registro o conexión de usuario (*user log on*), se requiere el identificaron usuario (ID) y una contraseña (*password*). El sistema permitirá la conexión del usuario sólo si el sistema conoce el indicador del usuario y el usuario conoce la contraseña asociada por el sistema a ese identificado. Este sistema ID/*password* es un método notablemente poco fiable para proporcionar control de acceso a los usuarios. Los usuarios pueden olvidar sus contraseñas y accidentalmente o de forma intencionada revelarlas. Los *hackers* han demostrado ser especialmente habilidosos en adivinar los identificadores y contraseñas de usuarios especiales, como el de control del sistema o el del personal de gestión de sistemas. Finalmente, el fichero que contiene identificadores y contraseñas siempre está sujeto a cualquier intento de penetración. Discutiremos las contramedidas en la Sección 16.3.

El control de acceso de usuarios en un entorno distribuido se puede llevar a cabo de forma centralizada o descentralizada. De una forma centralizada, la red proporciona un servicio de conexión, que determina a quién está permitido el uso de la red y con quién le está permitido conectarse.

El control acceso de usuarios descentralizado trata la red como un enlace de comunicación transparente, y el mecanismo de acceso habitual se realiza por parte del ordenador destino. Por supuesto, también deben considerarse todos los aspectos de seguridad relativos a la transmisión de contraseñas por la red.

En muchas redes, se puede utilizar el control de acceso en dos niveles. Los ordenadores de forma particular pueden proporcionar un servicio de conexión para proteger los recursos y aplicaciones específicas de ese ordenador. Adicionalmente, la red en su conjunto puede proporcionar una protección de acceso restringido únicamente a los usuarios autorizados. Esta organización en dos niveles es aconsejable en los casos habituales, en los cuales la red conecta diferentes ordenadores y únicamente proporciona mecanismos apropiados para el acceso entre un terminal y el ordenador. En una red de ordenadores más uniforme, se puede forzar la existencia de una política de acceso centralizado proporcionada por un centro de control de red.

CONTROL DE ACCESO ORIENTADO A LOS DATOS

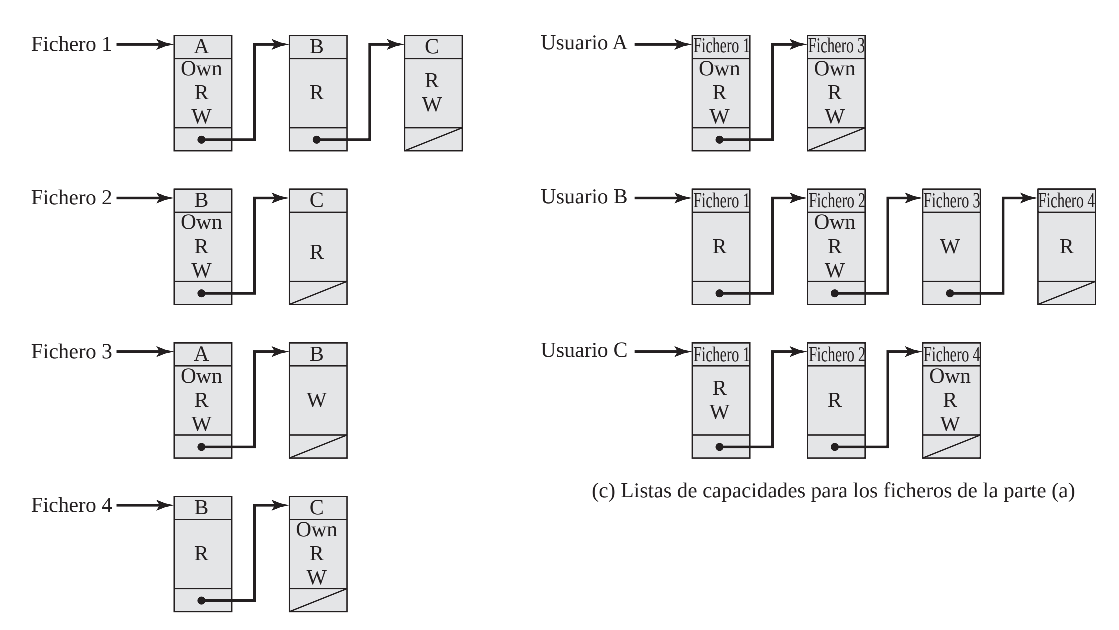
Una vez que se ha tenido éxito en la conexión al sistema, el usuario ha obtenido acceso a uno o más conjuntos de ordenadores y sus respectivas aplicaciones. Esto habitualmente no es suficiente para un sistema que incluye datos sensibles en sus bases de datos. Mediante un procedimiento de control acceso de usuario, se identifica a cada uno de los usuarios del sistema. Asociado con cada usuario, puede existir un perfil que especifica las operaciones permitidas en los accesos a ficheros. El sistema operativo puede entonces aplicar reglas que se basen en los perfiles de usuario. El sistema de gestión de bases de datos, sin embargo, debe controlar el acceso a determinados registros o incluso partes de los registros. Por ejemplo, se puede permitir que cualquier persona en la sección de administración obtenga un listado del personal de la compañía, pero sólo unos individuos determinados pueden tener acceso a la información de salarios. Esta cuestión es algo más que un simple nivel de detalle. Incluso en el caso en el cual un sistema operativo puede otorgar permisos a los usuarios para acceder a un fichero o utilizar una aplicación, a partir de lo cual no hay más controles de seguridad, un sistema de gestión de bases de datos debe tener en consideración los intentos de acceso de cada usuario. Esta decisión no dependerá únicamente de la identidad del usuario sino también de una parte específica de los datos a los cuales se va a acceder o incluso de la información ya divulgada al usuario.

Un modelo general para el control de acceso aplicado por un sistema de gestión de base datos o un sistema de ficheros es el denominado **matriz de acceso** (Figura 16.5a, basada en la figura de [SAND94]). Los elementos básicos del modelo son los siguientes:

- **Sujeto.** Un elemento capaz de acceder a los objetos. Realmente, el concepto de sujeto se asimila al de proceso. Todo usuario o aplicación realmente gana acceso a un objeto por medio de un proceso que representa al usuario o a la aplicación.
- **Objeto.** Todo elemento sobre el que se accede de forma controlada. Posibles ejemplos son ficheros, porciones de ficheros, programas, segmentos de memoria, objetos software (por ejemplo objetos Java).
- **Derecho de acceso.** La forma en la cual un objeto es accedido por un sujeto. Los diferentes ejemplos son lectura, escriturar, ejecución, y las funciones de objetos software.

	Fichero 1	Fichero 2	Fichero 3	Fichero 4	Account 1	Account 2
Usuario A	Own R W		Own R W		Información crédito	
Usuario B	R	Own R W	W	R	Información débito	Información crédito
Usuario C	R W	R		Own R W		Información débito

(a) Matriz de acceso



(b) Listas de control de acceso para los ficheros de la parte (a)

(c) Listas de capacidades para los ficheros de la parte (a)

Figura 16.5. Ejemplo de las estructuras de control de acceso.

Una de las dimensiones de la matriz consiste en los sujetos identificados que pueden intentar acceder a los datos. Habitualmente, esta lista consistirá en los usuarios o grupos de usuarios, aunque el control de acceso también se puede determinar por terminales, ordenadores y aplicaciones en lugar de o en conjunción con los usuarios. La otra dimensión indica los objetos sobre los cuales se realizan los accesos. A nivel de mayor detalle, los objetos pueden ser campos de datos individuales. Agrupaciones más generales, como registros, ficheros o una base datos completa, también pueden aparecer como objetos en la matriz. Cada entrada en la matriz indica los derechos de acceso que un sujeto tiene sobre dicho objeto.

En la práctica, una matriz de acceso es habitualmente bastante dispersa y se implementa mediante su descomposición de una de las dos maneras siguientes. La matriz se puede descomponer por columnas, definiendo **listas de control de acceso** (Figura 16.5b). De esta forma por cada objeto, hay una lista de control de acceso que muestra los usuarios y sus derechos de acceso. La listas de control acceso pueden contener una entrada por defecto, o pública. Esto permite que aquellos usuarios que no se encuentran indicados de forma explícita como propietarios de derechos especiales se les puedan asignar una serie de derechos por omisión. Los elementos de dichas listas pueden incluir usuarios individuales así como grupos de usuarios.

La descomposición por filas lleva a la definición de los **tickets de capacidades** (Figura 16.5c). Un ticket de capacidad específica objetos y operaciones autorizadas para un determinado usuario. El usuario tiene un número de tickets y puede encontrarse autorizado para cederlos o entregarlos a otros. Debido a que los tickets pueden encontrarse dispersos a lo largo del sistema, representa un problema de seguridad mayor que las listas de control acceso. En particular, un ticket no debe ser falsificable. Una forma de llevarlo a cabo es hacer que el sistema operativo se encargue de mantener los tickets de todos los usuarios. Dichos tickets pueden almacenarse en una región de memoria no accesible a los usuarios. Las consideraciones de red para el control de acceso orientado a datos son equiparables a aquellas para control acceso orientado a usuario. Si solo se permite el acceso a determinados usuarios y a ciertos elementos de datos, entonces puede resultar necesario el cifrado para proteger dichos elementos durante la transmisión a usuarios no autorizados. Habitualmente, el control de acceso a datos se encuentra descentralizado, esto es, controlando por sistemas de gestión de bases de datos basados en servidores. Si existe un servidor de base de datos de red dentro de una red, el control de acceso a datos se convierte en una función de red.

16.3. INTRUSOS

Uno de los dos peligros de seguridad más conocidos son los intrusos (el otro son los virus), generalmente denominados *hackers* o *crackers*. En un importante estudio preliminar sobre la intrusión en Anderson [ANDER80] se identifican tres clases de intrusos:

- **Enmascarado.** Un individuo que no está autorizado a utilizar un ordenador y que penetra en los controles de acceso del sistema para aprovecharse de una cuenta de usuario legítimo.
- **Trasgresor.** Un usuario legítimo que accede a datos, programas, o recursos para los cuales dicho acceso no está autorizado, o estando autorizado para dicho acceso utilizar sus privilegios de forma maliciosa.
- **Usuario clandestino.** Un usuario que sobrepasa el control de supervisión del sistema y usa dicho control para evadir la auditoría y el control de acceso o para suprimir la recogida de registros de acceso.

El intruso enmascarado se trata habitualmente de un usuario externo; el trasgresor generalmente es interno; el usuario clandestino puede ser indistintamente interno o externo.

Los ataques de los intrusos varían desde benigno hasta serio. En el lado benigno de la escala, se encuentran personas que simplemente desean explorar redes y saber qué hay ahí fuera. En el lado serio se encuentran los individuos que intentan leer datos privilegiados, realizar modificaciones no autorizadas a dichos datos o destruir un sistema.

TÉCNICAS DE INTRUSIÓN

El objetivo de un intruso es ganar acceso a un sistema o incrementar el rango de sus privilegios de acceso a dicho sistema. Generalmente, esto requiere que el intruso consiga información que debiera encontrarse protegida. En muchos casos, esta información se encuentra en forma de una contraseña de usuario. Con el conocimiento de la contraseña de otro usuario, un intruso puede acceder al sistema y utilizar todo los privilegios acordes al usuario legítimo.

Habitualmente, un sistema debe mantener un fichero que asocia las contraseñas con cada usuario autorizado. Si dicho fichero se encuentra almacenado sin ninguna protección, es fácil conseguir el acceso a él y leer las contraseñas. El fichero de contraseñas se puede proteger de formas diferentes:

- **Cifrado unidireccional.** El sistema almacena únicamente una forma cifrada de la contraseña de usuario. Cuando el usuarios presenta una contraseña, el sistema la cifra y la compara con el valor que tiene almacenado. En la práctica, el sistema habitualmente realiza una transformación unidireccional (no reversible) en la cual la contraseña suele generar un código cifrado de longitud fija.
- **Control acceso.** El acceso al fichero que contiene las contraseñas se encuentra limitado a una o muy pocas cuentas.

Si se aplican una o ambas de estas contramedidas, se requiere que el intruso potencial invierta un esfuerzo extra para poder conocer las contraseñas. Basándose en la revisión de la literatura y en entrevistas con diferentes *crackers* de contraseñas, [ALVA90] informa de las siguientes técnicas para conocer las contraseñas:

1. Intentar las contraseñas por defecto utilizadas para las cuentas estándar que vienen creadas con el sistema. Muchos administradores no se preocupan en cambiar estos valores por defecto.
2. Ensayar de forma exhaustiva todas las contraseñas de pequeño tamaño (aquellas que van desde uno a tres caracteres).
3. Probar palabras en el diccionario del sistema o una lista de contraseñas habituales. Ejemplos de estas últimas se encuentran disponibles en foros de *hackers* en Internet.
4. Recolectar información de los usuarios, tales como sus nombres completos, el nombre de sus esposas e hijos, imágenes de sus oficinas, libros en el despacho relacionados con sus *hobbies*.
5. Intentar el número del teléfono del usuario, el número de la seguridad social y la dirección del domicilio.
6. Probar con todo los números de matrículas válidos en dicha región.
7. Utilización de troyanos (descrito en la Sección 16.4) para sobrepasar las restricciones de acceso.
8. Pinchar la línea entre un usuario remoto y el sistema destino.

Los primeros seis métodos son diferentes mecanismos para adivinar una contraseña. Si el intruso tiene que verificar todas estas hipótesis intentando acceder a sistema, resulta tedioso y fácilmente evi-

table. Por ejemplo, un sistema puede simplemente rechazar cualquier intento de conexión después de haber probado tres contraseñas diferentes, esto requiere que el intruso vuelva a establecer la conexión con el ordenador e intentarlo de nuevo. Bajo estas circunstancias, no es práctico probar nada más que un puñado de contraseñas. Sin embargo, los intrusos no suelen probar estos mecanismos tan burdos. Por ejemplo, si el intruso tiene acceso al sistema con un nivel de privilegios muy bajo a un fichero de contraseñas que se encuentre cifrado, la estrategia sería capturar dicho fichero y utilizar el mecanismo de cifrado de ese sistema a voluntad hasta que se encuentre una contraseña válida que proporcione un nivel de privilegios más alto.

Los ataques por medio de la adivinación de contraseñas son habituales, e incluso altamente efectivos, sobre todo cuando se pueden evaluar un gran número de hipótesis de forma automática y verificar su validez o no, sin que el proceso de prueba puede detectarse. Más adelante en esta sección diremos más acerca de los ataques por adivinación de contraseñas.

El séptimo método indicado anteriormente, el troyano o caballo de Troya, puede ser particularmente difícil de contrarrestar. Un ejemplo de un programa que puede superar los controles de acceso se encuentra mencionado en [ALVA90]. Un usuario con pocos privilegios desarrolla un programa que es un juego e invita al administrador del sistema a utilizarlo en su tiempo libre. El programa verdaderamente es un juego, pero por detrás también contiene un código que realiza una copia del fichero de contraseñas, el cual no se encuentra cifrado pero si bajo protección de acceso. Dicho fichero se copia a un fichero de usuario. Debido a que el juego se está ejecutando bajo el modo privilegiado del administrador, es posible que tenga acceso al fichero de contraseñas.

El octavo ataque de la lista, pinchar o escuchar la línea, es una cuestión de seguridad física. Puede evitarse por medio de técnicas de cifrado en los enlaces.

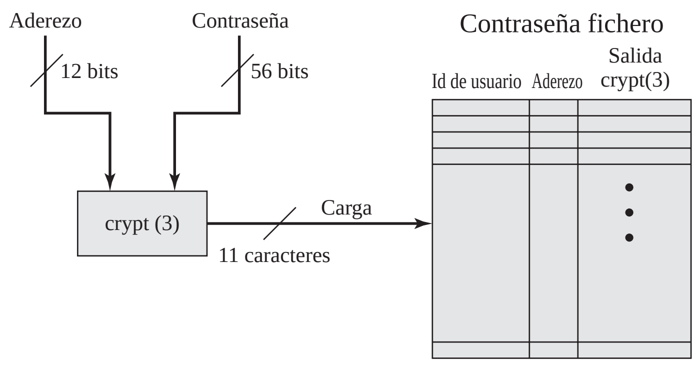
A continuación vamos a repasar las dos principales contramedidas: prevención y detección. La prevención es el reto de la seguridad informática y una dura batalla desde siempre. La dificultad estriba en el hecho de que el defensor debe guarecerse de todos los posibles ataques, mientras que el atacante tiene la libertad de intentar encontrar el punto más débil en la cadena defensiva y atacar en dicho punto. La detección se centra en el reconocimiento de los ataques, antes o justo después de que tengan éxito.

PROTECCIÓN DE CONTRASEÑAS

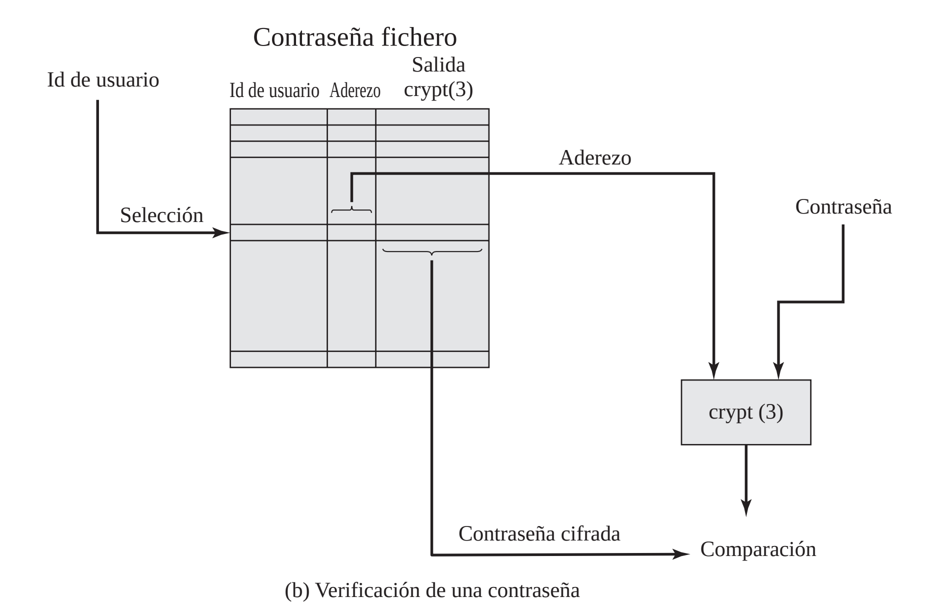
La primera línea de defensa contra los intrusos es el sistema de contraseñas. Virtualmente un acceso a un sistema multiusuario requiere que el usuario proporcione no sólo un nombre o identificador (ID) sino también una contraseña. La contraseña sirve para autentificar el identificador de aquel que se está conectando al sistema. El identificador proporciona seguridad de la siguiente manera:

- El identificador determina si el usuario está autorizado a conseguir el acceso al sistema. En algunos sistemas, sólo a aquellos que tienen un identificador ya registrado el sistema les permite conseguir el acceso.
- El identificador determina los privilegios asociados al usuario. Muy pocos usuarios pueden tener un estatus de supervisor o superusuario que les permite leer ficheros y realizar tareas que están especialmente protegidas por el sistema operativo. Algunos sistemas disponen de cuentas anónimas o de invitados, y los usuarios de dichas cuentas tienen unos privilegios más limitados que cualquiera de nosotros.
- El identificador también se utiliza para un control de acceso discrecional. Por ejemplo indicando la lista de los identificadores de otros usuarios, un usuario puede otorgar permisos para que estos puedan leer ficheros que le pertenecen a él.

Vulnerabilidad de las contraseñas. Para comprender la naturaleza del ataque, consideremos un esquema ampliamente difundido en los sistemas UNIX, en los cuales las contraseñas nunca se encuentran almacenadas en claro. En lugar de eso, se utiliza el siguiente procedimiento (Figura 16.6a). Cada usuario selecciona una contraseña de hasta seis caracteres imprimibles de longitud. Dicha contraseña se convierte en un valor de 56-bits (utilizando un código ASCII de 7-bits) que vale como entrada a una rutina de cifrado. La rutina de cifrado, conocida como crypt(3), se basa en DES (*Data Encryption Standard*), descrito en el Apéndice 16A. El algoritmo DES se modifica utilizando un valor de 12 bits de «aderezo». Típicamente, dicho valor se encuentra relacionado con el momento en el cual la contraseña se le asignó a dicho usuario. Al algoritmo DES modificado se le pasa como entrada un bloque de 64 bits a cero. La salida del algoritmo sirve a su vez de entrada para un segundo cifrado. Dicho proceso se repite un total de 25 veces. Resulta una salida final de 64 bits que se traduce en una secuencia de once caracteres. Esta contraseña de texto cifrado se almacena, junto con una copia en claro de valor de aderezo, en el fichero de contraseñas para el correspondiente identificador de usuario.



(a) Carga de una nueva contraseña



(b) Verificación de una contraseña

Figura 16.6. El esquema de contraseñas de UNIX.

El valor aderezo vale para tres fines diferentes:

- Evita que si hay contraseñas duplicadas éstas sean visibles en el fichero de contraseñas. Incluso si los usuarios eligen la misma contraseña, estas contraseñas habrá sido asignadas en diferentes instantes de tiempo. Por tanto, la contraseña «extendida» de ambos usuarios será diferente.
- Incrementa de forma efectiva la longitud de la contraseña sin requerir que el usuario recuerde dos caracteres adicionales. Por tanto, el número de posibles contraseñas se incrementa por un factor de 4096, incrementando la dificultad para adivinar dicha contraseña.
- Evita la utilización de implementaciones hardware de DES, que reduce el riesgo de ataques por fuerza bruta.

Cuando un usuario intenta acceder a un sistema UNIX, el usuario proporciona un identificador y una contraseña. El sistema operativo utiliza el identificador para indexar en el fichero de contraseñas y recuperar el valor de aderezo y la contraseña cifrada, el primero junto con la contraseña proporcionada por el usuario se utilizan como entrada a la rutina de cifrado. Si el resultado coincide con el valor almacenando, se acepta la contraseña.

La rutina de cifrado se diseñó para desalentar los intentos de adivinación de las contraseñas. Las implementaciones por software de DES son lentas comparadas con las versiones hardware, y la utilización de 25 iteraciones multiplica el tiempo necesario por 25. Sin embargo, desde el diseño original del algoritmo, han ocurrido dos cambios. El primero es que se han diseñado nuevas implementaciones del algoritmo con la consecuente mejora de prestaciones. Por ejemplo el gusano (*worm*) de Internet era capaz de intentar contraseñas en línea a una tasa de varios centenares de contraseñas en un periodo de tiempo relativamente breve, utilizando un algoritmo de cifrado mucho más eficiente que el proporcionado de forma estándar en los sistemas UNIX que atacaba. En segundo lugar, las prestaciones del hardware continúan incrementándose, de forma que cualquier algoritmo software se ejecuta mucho más rápidamente.

Así pues, existen dos peligros en el esquema de contraseñas de UNIX. El primero es que un usuario puede conseguir el acceso a un sistema utilizando una cuenta de invitado o por cualquier otro mecanismo y ejecutar un programa de adivinación de contraseñas, denominado habitualmente *password cracker*, en dicha máquina. El atacante puede ser capaz de probar cientos o incluso miles de posibles contraseñas con un consumo de recursos mínimo. Además, si un oponente es capaz de obtener una copia del fichero de contraseñas, el programa *cracker* se puede ejecutar en otra máquina a voluntad. Esto permite al oponente ejecutar, probando varios millares de posibles contraseñas, en un periodo razonable.

Como ejemplo, en agosto de 1993 se informó en Internet de la ejecución de un *password cracker* [MADS93]. Utilizando un ordenador paralelo de la *Thinking Machines Corporation*, que alcanzaba un rendimiento de 1560 cifrado por segundo por unidad vectorial. Con cuatro unidades vectoriales por nodo (la configuración estándar) se alcanzan un total de 800.000 cifrados por segundo en una máquina de 128 nodos (que representa un tamaño modesto) y 6,4 millones de cifrados por segundo en una máquina de 1024 nodos.

Incluso con estas increíbles tasas en la prueba de contraseñas no resulta factible para un atacante utilizar una técnica simple de fuerza bruta para probar todas las posibles combinaciones de caracteres para descubrir una contraseña. En lugar de eso, los *password crackers* se basan en el hecho de que la mayoría de las personas eligen contraseñas que son fácilmente adivinables.

Algunos usuarios, a los cuales se les permite elegir su propia contraseña, seleccionan alguna absurdamente corta. El resultado de un estudio realizado por la universidad de Purdue se muestra en la Tabla 16.2. El estudio observaba los cambios de contraseñas en 54 máquinas, que representaban

Tabla 16.2. Longitud de las contraseñas.

Longitud	Número	Fracción sobre el total
1	55	0.004
2	87	0.006
3	212	0.02
4	449	0.03
5	1260	0.09
6	3035	0.22
7	2917	0.21
8	5772	0.42
Total	13.787	1.0

aproximadamente 7000 cuentas de usuarios. Casi un 3% de las contraseñas eran de tres caracteres o menos de longitud. Un atacante podría comenzar el ataque probando exhaustivamente todas las posibles contraseñas de longitud tres o inferior. Un remedio sencillo es que el sistema rechace automáticamente cualquier selección de contraseña menor de, pongamos, seis caracteres o incluso que requiera que todas las contraseñas sean exactamente de ocho caracteres de longitud. Muchos usuarios no se quejarían de esta restricción.

La longitud de la contraseña es sólo parte del problema. Mucha gente, cuando se les permite elegir su propia contraseña, seleccionan una palabra que es fácilmente adivinable, tal como su propio nombre, el nombre de una calle, una palabra común del diccionario y similares. Esto hace que el trabajo del *password cracker* sea más sencillo. El programa simplemente tiene que probar el fichero de contraseñas contra aquellas palabras que parezcan más probables. Debido a que mucha gente utiliza contraseñas fácilmente adivinables, esta estrategia puede tener éxito en prácticamente todos los sistemas.

Una demostración de la efectividad de este mecanismo de adivinación de contraseñas se encuentra recogido en [KLEI90]. El autor recogió varios ficheros de contraseñas UNIX de diferentes fuentes, que contenían cerca de 14.000 contraseñas cifradas. El resultado, que el propio autor denomina de escalofriante, se muestra en la Tabla 16.3. En general, aproximadamente un cuarto de las contraseñas se consiguen adivinar, utilizando la estrategia siguiente:

1. Se intenta el nombre de usuario, iniciales, nombre de la cuenta, y otra información personal relevante. En total, se intentaron 130 permutaciones para cada uno de los usuarios.
2. Se intentaron también varios diccionarios. El autor recopiló un diccionario de más de 60.000 palabras, incluyendo el diccionario sistema y otros tantos como los que se muestran en la tabla.
3. Se intentaron varias permutaciones de las palabras del paso 2. Estas incluyen poner la primera letra en mayúsculas o un carácter de control, poner toda la palabra en mayúsculas, invertir la palabra, cambiar la letra «o» por el dígito «0», y demás. Estas permutaciones añadieron un millón de palabras a la lista.
4. Se consideraron otras combinaciones de mayúsculas y minúsculas de las palabras del paso 2 que no se habían considerado en el paso 3. Esto añadió casi 2 millones de palabras más a la lista.

Tabla 16.3. Contraseñas descifradas de un conjunto de entrenamiento de 13797 cuentas [KLEI90].

Tipo de contraseña	Tamaño de la búsqueda	Número de coincidencias	Porcentaje de las contraseñas coincidentes
Nombre de usuario/cuenta	130	368	2.7%
Secuencias de caracteres	866	22	0.2%
Números	427	9	0.1%
Chino	392	56	0.4%
Nombres de lugares	628	82	0.6%
Nombres comunes	2239	548	4.0%
Nombres de mujer	4280	161	1.2%
Nombres de hombre	2866	140	1.0%
Nombres poco habituales	4955	130	0.9%
Mitos y leyendas	1246	66	0.5%
Shakespeare	473	11	0.1%
Términos deportivos	238	32	0.2%
Ciencia ficción	691	59	0.4%
Películas y actores	99	12	0.1%
Dibujos animados	92	9	0.1%
Gente famosa	290	55	0.4%
Frases y términos	933	253	1.8%
Apellidos	33	9	0.1%
Biología	58	1	0.0%
Diccionario de sistema	19.683	1027	7.4%
Nombres de máquinas	9018	132	1.0%
Términos mnemotécnicos	14	2	0.0%
Biblia del rey Jaime	7525	83	0.6%
Palabras variadas	3212	54	0.4%
Palabras hebreas	56	0	0.0%
A esteroides	2407	19	0.1%
TOTAL	62.727	3340	24.2%

De esta forma, la prueba incluía un conjunto de palabras cercano a los 3 millones. Utilizando la implementación más rápida de *Thinking Machines* que hemos comentado anteriormente, el tiempo para cifrar todas estas palabras para todos los posibles valores de aderezo está por debajo de una hora. Hay que tener en cuenta que toda esta búsqueda puede producir una tasa de acierto de aproximadamente 25%, considerando que simplemente un acierto puede permitir ganar un amplio rango de privilegios en un sistema.

De forma más reciente, [PERR03] recoge una nueva estrategia que se aprovecha del continuo incremento de la velocidad de los procesadores y de la capacidad de almacenamiento. Bajo esta estrate-

gia, un gran número de posibles contraseñas con valores de aderezo alternativos se encuentran ya precomputadas y almacenadas en una tabla que puede ser consultada en el momento de descifrar las contraseñas. Este informe muestra la factibilidad de atacar un esquema de contraseñas UNIX.

Actualmente, muchos UNIX y la mayoría de distribuidores Linux ofrecen alternativas más seguras a la estrategia de la utilización únicamente de *crypt(3)*.

Control de acceso. Una forma de protegerse de los ataques de contraseñas es denegar el acceso al oponente al fichero de contraseñas. Si la parte del fichero de contraseñas cifradas se encuentra accesible sólo para los usuarios con el nivel de privilegios adecuados, el oponente no podrá leerlo sin conocer previamente la contraseña del usuario privilegiado. [SPAF92] remarca diferentes fallos en esta estrategia:

- Muchos sistemas, incluyendo la mayoría de sistemas UNIX, son susceptibles a intromisiones de una forma que no venga anticipada. Una vez que el atacante ha conseguido acceder de alguna forma al sistema, puede tener una lista de contraseñas de forma que puede utilizar diferentes cuentas a lo largo de distintas sesiones de conexión para reducir el riesgo de ser detectado. O, un usuario que ya dispone de una cuenta puede desear utilizar otra para acceder a datos privilegiados o sabotear el sistema.
- Un accidente en la protección puede hacer que el fichero de contraseñas sea legible, comprometiendo de esta forma todas las cuentas.
- Algunos de los usuarios pueden tener cuentas en otras máquinas bajo otros dominios de protección, y en algunos casos utilizar la misma contraseña. Por tanto, si las contraseñas pueden leerse por cualquiera en otra máquina, una máquina diferente puede encontrarse comprometida bajo estas circunstancias.

De esta forma, una estrategia más efectiva sería obligar a los usuarios a asignar contraseñas que sean difíciles de adivinar.

ESTRATEGIAS DE SELECCIÓN DE CONTRASEÑAS

La lección aprendida de los dos experimentos que acabamos de describir (Tablas 16.2 y 16.3) es que, dejada a su propia decisión, muchos usuarios eligen una contraseña que es demasiado corta o demasiado fácil de adivinar. En el otro extremo, si a los usuarios se les asignan contraseñas consistentes en ocho caracteres imprimibles seleccionados de forma aleatoria, la adivinación de las contraseñas es efectivamente imposible. Pero también resultaría imposible para la mayoría de los usuarios recordar sus propias contraseñas. Afortunadamente, incluso si limitamos el universo de contraseñas a las cadenas de caracteres que es razonable memorizar, el espacio de dicho universo es aún demasiado grande para permitir la adivinación de contraseñas de forma práctica. El objetivo, entonces, es eliminar las contraseñas adivinables mientras que permitimos que los usuarios seleccionen unas que sean fáciles de memorizar. Para ello se utilizan cuatro técnicas básicas:

- Educación de los usuarios.
- Contraseñas generadas por el ordenador.
- Verificación reactiva de las contraseñas.
- Verificación proactiva de las contraseñas.

A los usuarios se les puede decir la importancia de utilizar contraseñas difíciles de adivinar y se les pueden proporcionar referencias de cómo seleccionar contraseñas más fuertes. Esta estrategia de

educación de los usuarios no tiene muchas posibilidades de éxito en la mayoría de instalaciones, particularmente donde hay una amplia población de usuarios o muchos cambios de los mismos. Muchos usuarios simplemente ignoran estas recomendaciones. Otros puede ser que no sean buenos jueces de la fortaleza de la contraseña. Por ejemplo, muchos usuarios (erróneamente) creen que dar la vuelta a una palabra o poner en mayúsculas la última letra hace la contraseña imposible de adivinar.

Las **contraseñas generadas por ordenador** también tienen sus problemas. Si la naturaleza de la contraseña es demasiado aleatoria, los usuarios no la recordarán. Incluso si la contraseña es pronunciable, los usuarios pueden tener problemas en recordarla y sentirse tentados a tenerla escrita. En general, los esquemas de contraseñas generadas por ordenador tienen un historial de escasa aceptación por parte de los usuarios. FIPS PUBS 181 define uno de los generadores de contraseñas automáticos mejor diseñados. Este estándar incluye no sólo la descripción de la estrategia así como un listado completo del código fuente en C del algoritmo. El algoritmo genera palabras a partir de sílabas pronunciables que se concatena para formar una palabra. Un generador aleatorio de números produce un flujo de caracteres aleatorios usado para construir las sílabas y las palabras.

La **verificación reactiva de las contraseñas** es una estrategia mediante la cual el sistema de forma periódica ejecuta su propio programa de adivinación para encontrar posibles contraseñas adivinables. El sistema cancela cualquier contraseña que resulta adivinada y notifica al usuario. Esta táctica tiene varias desventajas. En primer lugar, requiere un uso intensivo de los recursos si se quiere realizar bien el trabajo. Debido a que un determinado oponente que es capaz de robar un fichero de contraseñas puede dedicar todo el tiempo de su CPU a la tarea durante horas o incluso días, una verificación reactiva de las contraseñas siempre tendrá esta desventaja. Además, todas las contraseñas que sean frágiles permanecerán vulnerables hasta el verificador de contraseñas las encuentre.

La alternativa más prometedora para mejorar el sistema de seguridad de contraseñas es la **verificación proactiva de las contraseñas**. En este esquema, a los usuarios se les permite seleccionar su propia contraseña. Sin embargo, en el momento de la selección, el sistema prueba a ver si la contraseña está permitida, y si no es así, la rechaza. Estas variaciones se basan en la filosofía de que, con suficiente supervisión por parte del sistema, un usuario puede seleccionar una contraseña que es posible memorizar de un espacio amplio de contraseñas que no son probablemente fáciles de adivinar en un ataque basado en el diccionario.

El truco para implantar un verificador de contraseñas proactivo es encontrar el equilibrio entre la aceptación por parte del usuario y la fuerza de la contraseña. Si el sistema rechaza demasiadas palabras, los usuarios se quejarán de que es muy difícil encontrar una contraseña. Si el sistema utiliza un algoritmo sencillo para definir qué es aceptable y qué no, esto podría proporcionar pistas para que los *password crackers* refinasen sus técnicas para adivinar las contraseñas. En el resto de esta subsección, vamos a repasar diferentes estrategias para realizar esta validación de contraseñas de forma proactiva.

La primera estrategia es un sistema sencillo de reglas. Por ejemplo, se pueden definir la siguientes reglas:

- Todas las contraseñas deben tener al menos ocho caracteres de longitud.
- En los primeros ocho caracteres, las contraseñas deben incluir al menos uno en mayúsculas, otro en minúsculas, un dígito y un signo de puntuación.

Estas reglas deben cumplirse mediante el aviso correspondiente al usuario y su validación. A pesar de que esta estrategia es mejor que simplemente indicar estas recomendaciones a los usuarios, puede no ser suficiente para defenderse de los ataques de adivinación de contraseñas. Este esquema puede alertar a los diferentes programas sobre cuáles son las contraseñas que no se tienen que probar pero hacen aun posible la adivinación del resto de ellas.

Otra posibilidad es simplemente recopilar un gran diccionario de lo que posiblemente sean malas contraseñas. Cuando el usuario selecciona una contraseña, el sistema comprueba si no se encuentra en la lista de palabras rechazadas. Existen dos problemas con esta estrategia:

- **El espacio.** El tamaño del diccionario debe ser muy grande para poder ser eficaz. Por ejemplo, el diccionario utilizado para el estudio de la universidad de Purdue [SPAF92] ocupa más de 30 Mbytes de almacenamiento.
- **El tiempo.** El tiempo que se requiere para buscar en un gran diccionario puede ser también excesivo. Adicionalmente, para verificar todas la permutaciones posibles de las palabras del diccionario, o bien se añaden esas palabras al diccionario, haciéndolo verdaderamente grande, o las búsquedas deben implicar un procesamiento considerable.

DETECCIÓN DE INTRUSOS

Inevitablemente, el mejor sistema para la prevención de intrusos fallará. Una segunda línea de defensa para el sistema es la detección de intrusos, que ha sido uno de los principales focos de investigación de los últimos años. Este interés se encuentra motivado por diversas consideraciones, incluyendo las siguientes:

1. Si una intrusión se detecta suficientemente rápido, el intruso puede ser identificado y expulsado del sistema antes de que haga daño o llegue a comprometer algún dato. Incluso si la detección no es suficientemente rápida para expulsar a tiempo, cuanto antes se detecte esta intrusión, menor será el daño que se provoca y más rápidamente se puede llegar a conseguir la recuperación.
2. Un sistema de detección que resulte efectivo puede actuar de forma disuasoria, de forma que se previenen intrusiones.
3. La detección de intrusos permite la recolección de información sobre las técnicas de intrusión y pueden utilizarse para reforzar los servicios de prevención.

La detección de intrusos se basa en la suposición de que el comportamiento de un intruso difiere del de un usuario legítimo de forma que puede ser cuantificado. Por supuesto, no se puede esperar que haya una distinción clara y exacta entre un ataque de un intruso y la utilización habitual de los recursos por parte del usuario autorizado. En lugar de ello, debemos esperar que exista determinado solapamiento.

La Figura 16.7 sugiere, de una forma bastante abstracta, la naturaleza de la tarea con la que se encuentra el diseñador de un sistema de detección de intrusos. A pesar de que el comportamiento típico de un intruso difiere del comportamiento típico de un usuario autorizado, existe un solapamiento entre estos dos comportamientos. De esta forma, una interpretación amplia de lo que puede ser el comportamiento de un intruso, permite capturar a un mayor número de ellos, pero llevará también a un elevado número de «falsos positivos», o usuarios autorizados identificados como intrusos. Por tanto, existen elementos de compromiso y una experiencia en la práctica de la detección de intrusos.

En el Estudio de Anderson [ANDE80], se postulaba que uno puede, con una confianza razonable, distinguir entre un intruso enmascarado y un usuario legítimo. Los patrones de comportamiento de los usuarios legítimos pueden observarse a partir de los datos históricos, y se puede detectar cualquier desviación significativa de dichos patrones. Anderson sugiere que la tarea de detección de un trasgresor (un usuario legítimo actuando de una forma no autorizada) es más difícil, en lo referente a que la distinción entre comportamiento anormal y el comportamiento normal puede ser muy pequeña. Anderson concluye que dichas violaciones serían indetectables únicamente a partir de la búsqueda de un

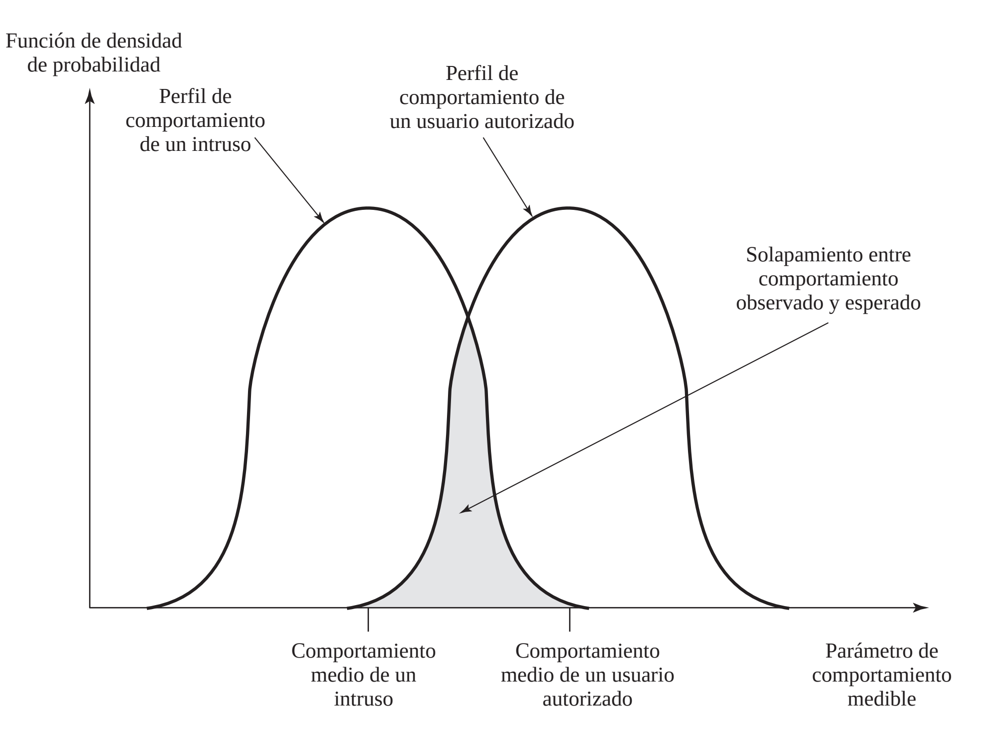


Figura 16.7. Perfiles de comportamiento de intrusos y usuarios autorizados.

comportamiento normal. Sin embargo, el comportamiento del trasgresor pueden en cualquier caso detectarse por medio de una definición inteligente de la clase de condiciones que sugieren un uso no autorizado. Para concluir, la detección del usuario clandestino va más allá del ámbito de las técnicas puramente remotas. Estas observaciones, que se hicieron en 1980, continúan siendo válidas hoy en día.

[PORR92] identifica las siguientes técnicas para detección de intrusos:

1. **Detección estadística de anomalías.** Implica la recolección de datos relativos al comportamiento de los usuarios legítimos durante un período de tiempo. Posteriormente se aplican una serie de tests estadísticos a un nuevo comportamiento que se quiere observar para determinar con un alto nivel de confianza si el comportamiento es o no el de un usuario legítimo.

   a) **Detección por umbral.** Esta estrategia implica definición de umbrales, independientes del usuario, para la frecuencia de determinados eventos.

   b) **Basado en el perfil.** Se desarrolla un perfil de actividad por cada uno de los usuarios que, se utiliza para detectar cambios en el comportamiento de cada una de las cuentas de forma individual.

2. **Detección basada en reglas.** Implica un intento de definir un conjunto de reglas que se puedan utilizar para decidir si un comportamiento dado es o no el de un intruso.

   a) **Detección de anomalías.** Reglas desarrolladas para detectar la desviación de los patrones de usos previos.

   b) **Identificación de penetración.** Un sistema experto que busca comportamiento sospechoso.

En esencia, las estrategias estadísticas intentan definir un comportamiento normal, o esperado, mientras que las técnicas basadas en reglas intentan definir un comportamiento sospechoso.

En relación a los tipos de atacantes mencionados anteriormente, la detección estadística de anomalías resulta efectiva para el caso de ataques enmascarados, que tienen pocas posibilidades de hacerse pasar por los patrones de comportamiento de las cuentas que se han apropiado. Por otro lado, estas técnicas no son capaces de tratar el caso de los trasgresores. Para este tipo de ataques, las estrategias basadas en reglas pueden identificar los eventos y las secuencias que, en su contexto, revelan una intromisión. En la práctica, un sistema puede exhibir una combinación de ambas estrategias para resultar efectivo contra un amplio abanico de ataques.

Una herramienta fundamental para detección de intrusos es el registro de auditoría. Se utilizan varios registros de las actividades que va realizando el usuario como entrada para los sistemas detección intrusos. Básicamente, dos de ellos son los más utilizados:

- **Registros de auditoría nativos.** Prácticamente todos los sistemas operativos multiusuario incluyen un software de auditoría con el fin de registrar la actividad de los usuarios. La ventaja de utilizar esta información es que no se requiere ningún software adicional para recoger estos datos. La desventaja es que los registros de auditoría nativos pueden no tener la información necesaria que puede requerirse o que no esté en el formato conveniente.
- **Registros de auditoría específicos para detección.** Se puede implementar una funcionalidad de recolección de datos que genere registros de auditoría que contienen información pensada únicamente para el sistema de detección de intrusos. Una ventaja de dicha estrategia es que puede realizarse de forma independiente del vendedor e implantarse en una amplia variedad de sistemas. La desventaja es que implica tener una sobrecarga extra, es decir, dos paquetes de auditoría ejecutándose en la misma máquina.

Un buen ejemplo de registros de auditoría específicos para detección son los desarrollados Dorothy Denning [DENN87]. Cada uno de los registros de auditoría tiene siguientes campos:

- **Sujeto.** Iniciador de la acción. Un sujeto es típicamente un terminal de usuario o puede también ser un proceso que actúa de parte de un usuario o grupo de usuarios. La actividad parte de una serie de mandatos lanzados por estos sujetos. Los sujetos pueden agruparse en diferentes clases de acceso, y dichas clases pueden estar solapadas.
- **Acción.** Operación realizada por el sujeto utilizando un objeto; por ejemplo acceso, lectura, operación de E/S, ejecución.
- **Objeto.** El receptor de las acciones. Los ejemplos incluyen ficheros, programas, mensajes, registros, terminales, impresoras y estructuras creadas por los usuarios o los programas. Cuando un sujeto es el receptor de una acción, por ejemplo un correo electrónico, entonces el sujeto se considera un objeto. Los sujetos pueden agruparse por tipos. La granularidad de los objetos puede variar por el tipo de objeto y por el entorno. Por ejemplo, las acciones o las bases de datos pueden auditarse para toda la base de datos en conjunto o a nivel de registro.
- **Condiciones de excepción.** Denota qué condiciones de excepción, si se dan, se lanzarían como respuesta.
- **Utilización de recursos.** Una lista de los elementos cuantitativos en los cuales los elementos calculan la cantidad de recursos utilizados (por ejemplo, número de líneas impresas o mostradas, número de registros leídos o escritos, tiempo de procesador, unidades de E/S utilizadas, tiempo de sesión transcurrido).
- **Sello de tiempo.** Un sello único de fecha y hora que identifica cuándo tuvo lugar esta acción.

Muchas de las operaciones de los usuarios constituyen varias acciones elementales. Por ejemplo, la copia de un fichero implica la ejecución de un programa de usuario, el cual incluye una verificación de acceso y el desarrollo de la copia propiamente dicha, más la lectura de un fichero, más la escritura de otro fichero. Consideremos el mandato

COPY JUEGO.EXE TO <Dir>JUEGO.EXE

lanzado por Pepe para copiar un fichero ejecutable JUEGO desde el directorio actual al directorio <Dir>. Se generarían los siguientes registros de auditoría:

Pepe	ejecución	<Dir>COPY.EXE	0	CPU=00002	11058721678
Pepe	lectura	<Pepe>JUEGO.EXE	0	RECORDS=0	11058721679
Pepe	ejecución	<Dir>JUEGO.EXE	wr-viol	RECORDS=0	11058721680

En este caso, la operación de copia se ha abortado debido a que Pepe no tiene permisos escritura en <Dir>.

La división de las operaciones de usuario en acciones elementales trae consigo tres ventajas:

1. Debido a que los objetos son las entidades del sistema bajo protección, la utilización de acciones elementales permite una auditoría de todo comportamiento que se refiere a un objeto. De esta forma, el sistema puede detectar intentos maliciosos de controles de acceso (notando esta anomalía en el número de la condición de excepción devuelto) y puede detectar las acciones maliciosas que han tenido éxito notando la anomalía en el conjunto de objetos accesibles por dicho sujeto.
2. Los registros de auditoría basados en un objeto sencillo y en una acción sencilla simplifican el modelo de implementación.
3. Debido a la estructura uniforme y sencilla de los registros de auditoría específicos para detección, es relativamente sencillo obtener esta información, o al menos parte de ella, proyectando directamente los registros de auditoría nativos actuales en los registros específicos para la detección.

16.4. SOFTWARE MALICIOSO

Quizá los tipos más sofisticados de amenazas para un sistema informático se encuentran presentes en los programas que explotan las vulnerabilidades de dicho sistema. En este contexto, nos centraremos en programas de aplicación así como programas de utilidad, tales como editores y compiladores. El término genérico de estas amenazas es software malicioso o *malware*. El *malware* es software diseñado para causar daño o utilizar recursos de un ordenador. Frecuentemente se encuentra escondido dentro un programa enmascarado como software legítimo. En algunos casos, se distribuye asimismo a otros ordenadores por medio del correo electrónico o de diskettes infectados.

Comenzamos esta sección con una visión general del espectro relativo a estas amenazas software. El resto de la sección se encuentra dedicado a los virus, en primer lugar veremos cuál es su naturaleza y las posibles medidas para evitarlos.

PROGRAMAS MALICIOSOS

La Figura 16.8 proporciona una taxonomía general del software malicioso. Estas amenazas se pueden dividir en dos diferentes categorías: aquellas que necesitan un programa anfitrión y aquellas que son independientes. Las primeras son esencialmente fragmentos de programa que no pueden existir de forma independiente sin una aplicación, utilidad o programa de sistema en particular. Las últimas son programas autónomos que pueden planificarse y ejecutarse por parte del sistema operativo.

También podemos realizar la diferencia entre los casos que no se pueden replicar y aquellos que sí. Los primeros son meros de programas que deben activarse cuando el programa anfitrión se ejecuta para realizar una función específica. Los segundos consisten o bien en un fragmento de programa (virus) o un programa independiente (gusano, zombie) que cuando se ejecutan, pueden producir una o más copias de sí mismo que se activarán posteriormente en el mismo o en otros sistemas.

A pesar de que la taxonomía de la Figura 16.8 puede ser útil para organizar la información que estamos tratando, no muestra todo el panorama. En particular, las bombas lógicas o los troyanos pueden ser parte de un virus o gusano.

Puerta secreta. Una puerta secreta es un punto de entrada secreto dentro de un programa que permite a alguien que conoce la existencia de dicha puerta secreta tener el acceso sin utilizar los procedimientos de acceso de seguridad estándar. Las puertas secretas se han utilizado de forma legítima durante muchos años por los programadores para depurar programas. Habitualmente cuando el programador está desarrollando una aplicación que tiene un procedimiento de autenticación, a una configuración relativamente larga, que requiere que el usuario introduzca muchos valores diferentes para ejecutar la aplicación. Para depurar el programa, el desarrollador puede querer ganar privilegios especiales o evitar toda la configuración y autenticación estándar. También puede querer asegurarse de que existe un método para activar el programa si en algún momento se da un problema con el procedimiento de autenticación que está incluido dentro de la aplicación. La puerta secreta es un código que reconoce determinadas secuencias especiales de entrada o que se dispara cuando se ejecuta por parte de determinado identificador de usuario o una secuencial de eventos poco habitual.

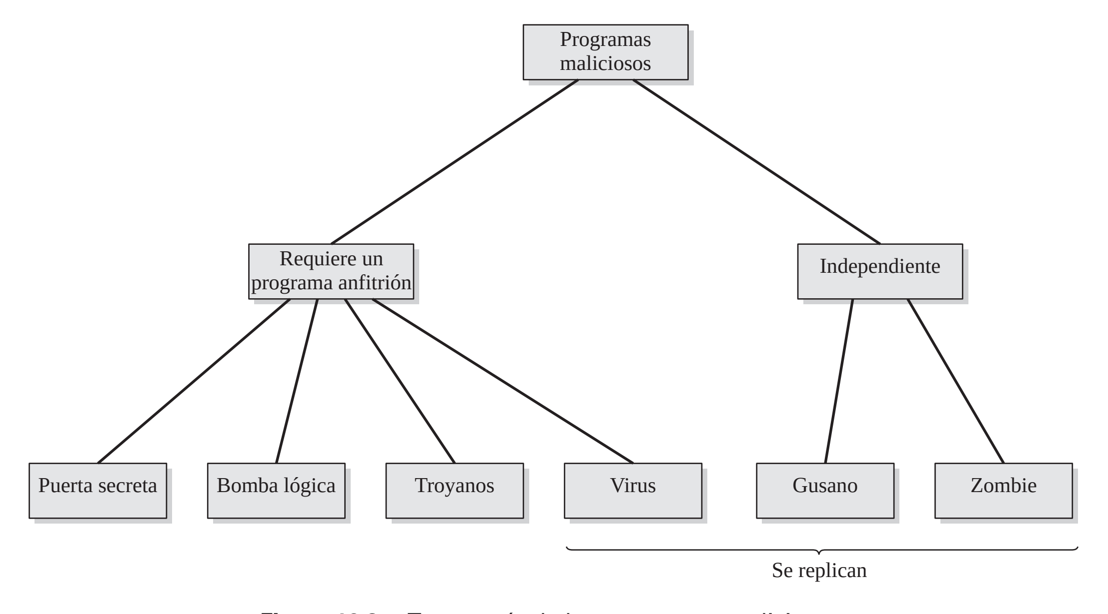


Figura 16.8. Taxonomía de los programas maliciosos.

Las puertas secretas pueden convertirse en amenazas cuando son utilizadas por programadores sin escrúpulos para ganar accesos no autorizados. La puerta secreta fue la idea básica de la vulnerabilidad retratada en la película Juegos de Guerra [COOP89]. Otro ejemplo es que, durante el desarrollo de Multics se realizaron una serie de pruebas de penetración por parte de un grupo denominado Air-Force «*tiger team*» (adversarios simulados). Una técnica empleada era enviar una actualización falsa del sistema operativo a un ordenador que estaba ejecutando Multics. La actualización contenida un troyano que se podría activar por medio de una puerta secreta que permitía al *tiger team* conseguir el acceso al sistema. La amenaza estaba tan bien implementada que el equipo de desarrolladores de Multics no fue capaz de encontrarlo, incluso después de haber sido informados de su presencia [ENGE80].

Es muy difícil implementar controles por parte del sistema operativo para las puertas secretas. Las medidas de seguridad se deben centrar en el desarrollo de los programas y las actividades relativas a la actualización del software.

Bomba lógica. Uno de los más viejos peligros, anterior incluso a los virus y los gusanos, son las bombas lógicas. Una bomba lógica es un código insertado dentro de un programa legítimo que explotará bajo ciertas condiciones. Los ejemplos de las diferentes condiciones que pueden activar una bomba lógica son la presencia o ausencia de determinados criterios, un día particular de la semana o una fecha, o un usuario en particular ejecutando una aplicación. Una vez activada, la bomba puede alterar o borrar datos o ficheros completos, causando que la máquina se detenga, o que se produzca algún daño. Por ejemplo, tuvo mucho impacto la utilización de una bomba lógica en el caso de Tim Lloyd, que fue condenado por poner una que costó a su empresa, Omega Engineering, más de 10 millones de dólares, echando por tierra la estrategia de crecimiento de la empresa, y causando el despido de 80 trabajadores [GAUD00]. En última instancia, Lloyd fue sentenciado a 41 meses de prisión y obligado a pagar 2 millones de dólares de indemnización.

Troyano. Un caballo de Troya o troyano es un programa útil, o aparentemente útil, o mandato que contiene un código oculto que, al invocarse, realiza una función no deseada o dañina.

Los programas troyanos se utilizan para realizar tareas de forma indirecta que el usuario no autorizado no podría realizar directamente. Por ejemplo, para ganar el acceso a determinados ficheros de otro usuario en un sistema compartido, un usuario puede crear un troyano que, cuando se ejecuta, cambie los permisos de los ficheros usados de forma que los ficheros sean accesibles posteriormente por cualquier usuario. El autor posteriormente puede inducir a otros usuarios a ejecutar el programa colocándolo en un directorio común y renombrándolo como si se tratase de una herramienta de utilidad. Un posible ejemplo sería un programa que produce un listado de los ficheros del usuario en un formato determinado. Después de que otro usuario utilice este programa, el autor puede tener acceso a la información de los ficheros del mismo. Un ejemplo de un programa troyano que puede ser difícil de detectar sería un compilador que haya sido modificado para insertar un código adicional en determinados programas cuando éstos se compilan, por ejemplo en el caso del programa *login* del sistema [THOM84]. El código puede crear una puerta secreta en un programa de *login* que permite al autor acceder al sistema utilizando una clave especial. Este troyano nunca será descubierto inspeccionando el código fuente del programa de *login*.

Otra motivación habitual de la utilización de troyanos es la destrucción de datos. El programa puede parecer que realiza una tarea útil (por ejemplo un programa calculadora), pero puede también ir borrando silenciosamente los ficheros del usuario. Por ejemplo, un ejecutivo de CBS fue víctima de un troyano que destruyo toda la información contenida en su ordenador [TIME90]. El troyano se presentaba como una rutina gráfica ofrecida en un foro.

Virus. Un virus es un programa que puede infectar otros programas modificándolos; las modificaciones incluyen la copia del programa virus, que puede a continuación infectar otros programas.

Los virus biológicos son pequeños fragmentos de código genético —ADN o ARN— que pueden tomar la maquinaria de una célula viva y modificarla para realizar miles de réplicas del virus original. Como su análogo biológico, un virus informático contiene un código de instrucciones que se encarga de realizar copias de sí mismo. Infectado un ordenador, un virus típico toma control del disco del sistema operativo del ordenador. Posteriormente, en el momento en que el ordenador infectado se pone en contacto con otro software éste quedará infectado, una nueva copia del virus pasa a este programa. De esta forma, la infección se puede expandir de ordenador en ordenador sin que los usuarios se percaten, bien por medio del intercambio de discos o enviándose programas de uno a otro a través de la red. En un entorno de red, la posibilidad de acceder a aplicaciones y servicios de sistema de otro ordenador proporciona una infraestructura perfecta para la dispersión de los virus.

Los virus se examinan con mayor detalle en la siguiente sección.

Gusano. Los gusanos utilizan las conexiones de red para expandirse de un sistema a otro. Una vez que se encuentran activos dentro de un sistema, un gusano de red se puede comportar como un un virus informático, puede implantar troyanos o realizar cualquier otro tipo de acciones destructivas.

Para replicarse a sí mismo, un gusano de red utiliza algún tipo de vehículo de comunicación. Por ejemplo:

- **Herramientas de correo electrónico.** El gusano envía por correo una copia de sí mismo a otros sistemas.
- **Capacidad ejecución remota.** El gusano ejecuta una copia de sí mismo en otro sistema.
- **Capacidad de conexión remota.** El gusano se conecta a un sistema remoto como un usuario y posteriormente utiliza los mandatos para copiarse a sí mismo de un sistema a otro.

La nueva copia del programa gusano se ejecuta entonces en el sistema remoto donde, junto con otras funciones que puede realizar en el sistema, continúa su dispersión de la misma forma.

Un gusano de red muestra las mismas características que un virus informático: una fase latente, una fase de preparación, una fase de activación y una fase de ejecución. La fase de propagación realmente realiza las siguiente acciones:

1. Búsqueda de otros sistemas a infectar examinando las tablas de servidores o repositorios similares de direcciones remotas.
2. Establecimiento de una conexión con el sistema remoto.
3. Copia de sí mismo al sistema remoto y ejecución.

Un gusano de red puede intentar determinar si el sistema ya había sido previamente infectado antes de copiarse a sí mismo en el sistema. En un sistema multiprogramado, puede disfrazar su presencia renombrándose como un proceso de sistema o utilizando cualquier otro nombre que no resulte sospechoso para el operador de sistema.

Como en el caso de los virus, los gusanos de red son difíciles de contrarrestar. Sin embargo, tanto las medidas de seguridad de red como las de seguridad de sistemas, si están debidamente diseñadas e implementadas, minimizan la amenaza de los gusanos.

Zombie. Un programa zombie toma el control de otro ordenador conectado a Internet y posteriormente utiliza el mismo para lanzar ataques que son difíciles de trazar como provenientes del creador del zombie. Los zombies se utilizan habitualmente para ataques de denegación de servicio, habitualmente contra sitios web que son sus objetivos. Un zombie se implanta en centenares de ordenadores

pertenecientes a terceras partes, que desconocen su existencia, y posteriormente utiliza todos estos puntos de acceso para derribar el sitio web en cuestión lanzando un tráfico de red intenso.

LA NATURALEZA DE LOS VIRUS

Un virus puede realizar lo mismo que cualquier otro programa. La única diferencia es que se inserta él mismo dentro de otro programa y se ejecuta secretamente cuando el programa anfitrión se va a ejecutar. Una vez que el virus está en ejecución, puede realizar cualquier función que esté permitida con los privilegios del usuario en cuestión, tal como borrar ficheros y programas.

Durante su tiempo de vida, un virus típico pasa por las siguientes cuatro etapas:

- **Fase latente.** El virus está dormido. El virus se activará finalmente cuando se dé un evento, por ejemplo una fecha, la presencia de otro programa o ficheros, o que la capacidad del disco exceda de un determinado límite. No todos los virus pasan por esta etapa.
- **Fase de propagación.** El virus inserta copias idénticas de sí mismo en otros programas o en ciertas áreas de disco. Cada programa infectado contendrá ahora una copia del virus, que a su vez entrará en la fase de propagación.
- **Fase de activación.** El virus se activa para realizar la función para la cual fue concebido. Con el caso de la fase latente, la fase de activación se puede lanzar por una amplia gama de eventos del sistema, incluyendo un contador del número de veces que esta copia del virus se ha copiado a sí mismo.
- **Fase de ejecución.** La función en cuestión se realiza. La función puede variar desde un mensaje inofensivo en la pantalla, hasta la dañina destrucción de programas y ficheros de datos.

La mayoría de los virus llevan a cabo su trabajo de una manera específica de un sistema operativo en particular, y en algunos casos, específica también para la plataforma hardware. Por tanto, están diseñados para aprovecharse de los detalles y debilidades de determinados sistemas en concreto.

TIPOS DE VIRUS

Ha habido una continua carrera entre los diseñadores de virus y los diseñadores de software antivirus desde que los virus aparecieron por primera vez. A medida que se iban desarrollando medidas contra los tipos de virus existentes, se iban desarrollando nuevos virus. [STEP93] sugiere las siguientes categorías entre las más significativas de los tipos de virus:

- **Virus parásito.** Forma tradicional y asimismo la más habitual de un virus. Un virus parásito se inserta él mismo dentro de ficheros ejecutables y se replica, cuando el programa infectado se encuentra en ejecución, buscando otros ficheros ejecutables que infectar.
- **Virus residente en memoria.** Infecta la memoria principal como parte del programa del sistema residente. Desde ese punto, el virus infectar todo programa que se ejecuta.
- **Virus en el sector de arranque.** Infecta el sector de arranque maestro (*master boot record*) y se dispersan cuando el sistema arranca desde el disco que contiene el virus.
- **Virus oculto.** Una forma de virus diseñada expresamente para esconderse del software de detección de los antivirus.
- **Virus polimórfico.** Un virus que muta con cada infección, haciendo que la detección por medio de la «firma» del virus sea imposible.

Un ejemplo de un **virus oculto** es aquel que utiliza compresión de forma que el programa infectado tenga exactamente la misma longitud que la versión no infectada. Existen técnicas mucho más sofisticadas. Por ejemplo, un virus puede colocar una lógica de interceptación de las rutinas de E/S del disco, de forma que si se intenta leer las partes sospechosas del disco utilizando estas rutinas, el virus presentará la versión original, no infectada del programa. De esta forma, la ocultación no es un término que se aplique a los virus como tal sino, más bien, es una técnica que utilizan para evitar la detección.

Un **virus polimórfico** crea copias durante la fase de replicación que son funcionalmente equivalentes pero que representan diferentes patrones de bits. Como el caso de los virus ocultos, el propósito es derrotar a los programas que se encargan de buscar virus. En este caso, la «firma» del virus variará con cada copia. Para conseguir esta variación, el virus puede insertar de forma aleatoria instrucciones superfluas o intercambiar el orden de instrucciones que sean independientes. Una estrategia más efectiva es el uso de cifrado. Una parte del virus, generalmente denominada motor de mutación, creará una clave de cifrado aleatoria para cifrar el resto del virus. La clave se almacena junto con el virus, y el motor de mutación se alterará a sí mismo. Cuando se invoca el programa infectado, el virus utiliza la clave almacenada para descifrar el virus. Cuando el virus se replica, se selecciona una nueva clave aleatoria.

Otra arma dentro del repertorio de los diseñadores de virus es el denominado *virus-creation toolkit*. Estas herramientas permiten a alguien relativamente novato crear rápidamente gran número de virus diferentes. A pesar de que los virus creados con estas herramientas tienden a ser menos sofisticados que los virus diseñados desde cero, la gran cantidad de nuevos virus que se generan representa un grave problema para los esquemas de los antivirus.

Otra herramienta utilizada por los diseñadores de virus es el foro «*virus exchange bulletin board*» Un gran número de estos foros ha sido expulsado [ADAM92] de los Estados Unidos así como de otros países. Estos foros ofrecen copias de virus que pueden descargarse, así como trucos en la creación de los virus.

VIRUS BASADOS EN MACROS

En los últimos años, el número de virus que se han encontrado en determinadas empresas se ha incrementado dramáticamente. En gran parte este incremento se debe a la proliferación de un tipo de virus basado en macros. Los virus basados en macros son especialmente peligrosos debido a una serie de razones:

1. Un virus basado en macros es independiente de la plataforma. Prácticamente todos los virus basados en macros infectan documentos de Microsoft Word. Cualquier plataforma hardware o sistema operativo que soporte Word puede resultar infectado.
2. Los virus basados en macros infectan documentos, no fragmentos de código ejecutables. La mayoría de la información introducida dentro de un ordenador está en forma de documentos más que de programas.
3. Los virus basados en macros tienen una distribución más sencilla. Un mecanismo relativamente común es por medio del correo electrónico.

Los virus basados en macros se aprovechan de unas características presentes en Word y otras aplicaciones de ofimática por ejemplo Microsoft Excel, denominadas macros. En esencia, una macro es un programa ejecutable embebido en un documento de un procesador de textos o en otro tipo de ficheros. Normalmente, los usuarios utilizan las macros para automatizar determinadas tareas y de esta forma ahorrar pulsaciones de teclado. El lenguaje de las macros es habitualmente alguna forma de lenguaje de programación Basic. Un usuario puede definir una secuencia de pulsaciones en una ma-

cro y configurar para que esta macro se invoque cuando se pulse una tecla de función o una combinación corta de teclas.

Lo que permite la creación de un virus basado en macros son las macros autoejecutables. Éstas son macros que se invocan automáticamente, sin intervención explícita del usuario. Eventos autoejecutables críticos son abrir un fichero, cerrar un fichero o comenzar una aplicación. Una vez que la macro se ejecuta, se puede copiar a sí misma a otros documentos, borrar ficheros y causar otro tipo de daños al sistema. En Microsoft Word, hay tres tipos de macros diferentes de autoejecución:

- **Autoejecutable.** Si se encuentra una macro llamada AutoExec en el fichero de plantilla «normal.dot» o en una plantilla global almacenada en el directorio de configuración de Word, ésta se ejecutará siempre que se arranca Word.
- **Automacro.** Una automacro se ejecuta cuando ocurre un evento predefinido, tal y como la apertura o cierre de un documento, la creación de un nuevo documento o salir de Word.
- **Macro de mandato.** Si una macro en un fichero global de macros o una macro incluida dentro un documento tiene el nombre de un mandato existente en Word, se ejecutará siempre que se invoque dicho mandato (por ejemplo FileSave).

Una técnica habitual para difundir los virus basados en macros es la siguiente. Se añade una automacro o una macro de mandato en un documento Word que se introduce dentro del sistema por correo electrónico o transferencia de ficheros. En algún momento después de que el documento se ha abierto, la macro se ejecuta. Esta macro se copia a sí misma a un fichero global de macros. Cuando se abre la siguiente sesión de Word, la macro global infectada se encuentra activa. Cuando ésta se ejecuta, se puede replicar a sí misma y causar el daño.

Las sucesivas versiones de Word proporcionan una protección cada vez mayor contra este tipo de virus. Por ejemplo, Microsoft ofrece una herramienta opcional llamada *Macro Virus Protection* que detecta ficheros sospechosos de Word y alerta al usuario del peligro potencial de abrir ficheros con macros. Varios vendedores de productos antivirus han desarrollado también herramientas para detectar y eliminar virus basados en macros. Como en el caso de otros tipos de virus, la competición continúa en el campo de los virus basados en macros.

ESTRATEGIAS DE LOS ANTIVIRUS

La solución ideal para la amenaza de los virus es la prevención: no permitiendo que un virus entre en el sistema, en primer lugar. Este objetivo es, en general imposible de conseguir, sin embargo la prevención puede reducir el número de ataques de virus que tengan éxito. La siguiente opción es conseguir realizar lo siguiente:

- **Detección.** Una vez que ha ocurrido una infección, determinar que, efectivamente, ha habido una infección y localizar el virus.
- **Identificación.** Una vez que la detección se ha llevado acabo, identificar qué virus específico ha infectado el programa.
- **Eliminación.** Una vez que se ha identificado el virus en concreto, se eliminan todos los restos del virus del programa infectado y se recupera a su estado original. Eliminar el virus de todos los sistemas infectados hace que la infección no se pueda expandir más.

Si la detección tiene éxito pero no es posible ni la identificación ni la eliminación, entonces la alternativa consiste en deshacernos del programa infectado y volver a cargar una versión limpia desde la copia de *backup*.

Los avances en las tecnologías de los virus y los antivirus van mano a mano. Los primeros virus eran fragmentos de código relativamente simples y podrían identificarse y eliminase con paquetes de software de antivirus también relativamente simples. A medida que la carrera armamentista de los virus ha evolucionado, los virus y, necesariamente, el software de antivirus han crecido hasta convertirse en algo mucho más complejo y sofisticado. En la actualidad van apareciendo estrategias y paquetes de antivirus cada vez más sofisticados. En esta sección, vamos a remarcar dos de las estrategias más importantes.

Descifrado genérico. La tecnología de descifrado genérico, o *generic decryption* (GD), permite a los programas antivirus detectar fácilmente incluso a los virus polimórficos más complejos manteniendo unas velocidades de exploración altas [NACH97]. Recordemos que cuando un fichero que contiene un virus polimórfico se quiere ejecutar, el virus debe descifrarse a sí mismo para poder activarse. Con el objetivo de detectar esta estructura, todos los ficheros ejecutables se recorren con un escáner GD, que tiene los siguientes elementos:

- **Emulador de CPU.** Un ordenador virtual basado en software. Las instrucciones que se encuentran dentro de un fichero ejecutable se interpretan por parte del emulador, en lugar de ejecutarlas el procesador nativo. El emulador incluye una versión software de todos los registros y demás hardware del procesador, de forma que el procesador nativo no se ve afectado por los programas interpretados por el emulador.
- **Escáner de firma de virus.** Es un módulo que recorre el código a analizar buscando la firma de virus conocidos.
- **Módulo de control de la emulación.** Controla la ejecución del código a analizar.

Al comienzo de cada simulación, el emulador comienza a interpretar las instrucciones del código a analizar, de una en una. De esta forma, si el código incluye una rutina de descifrado que lo desvela y por tanto expone el virus, dicho código se interpreta. En efecto, el virus realiza el trabajo para el programa antivirus exponiendo el código del virus. Periódicamente, el módulo de control interrumpe a la interpretación para analizar el código en busca de firmas de virus.

Durante la interpretación, el código que se va analizando no puede causar daño alguno al entorno del ordenador personal, debido a que se encuentra interpretado en un entorno completamente controlado.

La consideración de diseño más difícil de un escáner GD es determinar durante cuánto tiempo se tiene que ejecutar una interpretación. Habitualmente es común que los elementos de virus se activan poco después de que el programa empiece a ejecutar, pero no es necesariamente así para todos los casos. Cuanto más tiempo se tome el escáner en emular un programa en particular, es más probable que encuentre virus que estén ocultos.

Sistema de inmunidad digital. El sistema de inmunidad digital es una inteligente estrategia para la protección contra virus desarrollada por IBM [WHIT99, KEPH97a, KEPH97b]. La motivación para dicho desarrollo fue la creciente amenaza de la propagación de virus basados en Internet. En primer lugar vamos a comentar un poco sobre esta amenaza y posteriormente resumiremos la estrategia de IBM.

Tradicionalmente, la amenaza de los virus se caracterizaba por la difusión relativamente lenta de nuevos virus y nuevas mutaciones. El software antivirus se actualizaba normalmente de forma mensual, y esto era suficiente para controlar el problema. Hasta finales de los años 90 Internet jugaba un papel relativamente pequeño en la difusión de los virus. Pero como [CHES97] remarca, ha habido dos tendencias en la tecnología de Internet que han incrementado su impacto en la propagación de virus:

- **Sistemas de correo integrado.** Sistemas tales como Lotus Notes o Microsoft Outlook hacen muy sencillo el envío de cualquier cosa a cualquier destinatario así como el trabajar con los objetos que han sido recibidos.
- **Sistemas de programas móviles.** Funcionalidades como Java o ActiveX permiten a los programas moverse por ellos mismos de un sistema a otro.

En respuesta a la amenaza planteada por estas nuevas funcionalidades de Internet, IBM ha desarrollado un prototipo para un sistema de inmunidad digital. Este sistema extiende el uso de la emulación de programas presentado en la subsección anterior y proporciona una emulación de propósito general y un sistema de detección de virus. El objetivo de este sistema es proporcionar un tiempo de respuesta rápido de forma que los virus puedan ser expulsados casi tan rápido como se introducen. Cuando un nuevo virus entra en una organización, el sistema de inmunidad automáticamente lo captura, lo analiza, añade mecanismos de detección y defensa contra él, lo elimina, y pasa información sobre dicho virus a los sistemas que ejecutan IBM Anti Virus de forman que pueda detectarse antes de que se permita su ejecución en cualquier otro lugar. La Figura 16.9 muestra los pasos típicos en una operación de un sistema de inmunidad digital:

1. Un programa de monitorización en cada PC utiliza una amplia gama de heurísticas basadas en el comportamiento del sistema, cambios sospechosos en los programas, firmas familiares para inferir que un virus se encuentra presente. El programa de monitorización envía una copia de cualquier programa que se piensa que puede estar infectado a una máquina de administración dentro de la organización.
2. La máquina de administración cifra la muestra y la envía a una máquina de análisis de virus central.
3. Esta máquina crea un entorno en el cual el programa infectado puede ejecutarse de forma segura para su análisis. Las técnicas utilizadas para ese fin incluyen la emulación, o la creación de un entorno protegido dentro del cual el programa sospechoso puede ejecutarse y monitorizarse. La máquina de análisis de virus finalmente produce un tratamiento para identificar y eliminar el virus.
4. El tratamiento resultante se envía de nuevo a la máquina de administración.
5. La máquina de administración reenvía el tratamiento al cliente infectado.
6. El tratamiento también se envía a otras máquinas clientes en la organización.
7. Los suscriptores a lo largo del planeta reciben actualizaciones regulares de los antivirus que les protegen de cualquier nuevo virus.

El éxito del sistema de inmunidad digital depende de la habilidad de la máquina de análisis de virus para detectar corrientes de virus nuevas e innovadoras. Por medio del análisis y la monitorización constante de virus encontrados en el exterior, es posible actualizar continuamente el software de inmunidad digital para mantenerlo en defensa contra cualquiera amenaza.

VIRUS POR CORREO

Los últimos desarrollos en el software malicioso son los virus por correo. Los primeros virus por correo que se difundieron rápidamente, por ejemplo Melissa, utilizaba las macros de Microsoft Word incluidas en un fichero adjunto. Si el receptor abría el fichero adjunto de un correo, la macro de Word se activaba. Entonces:

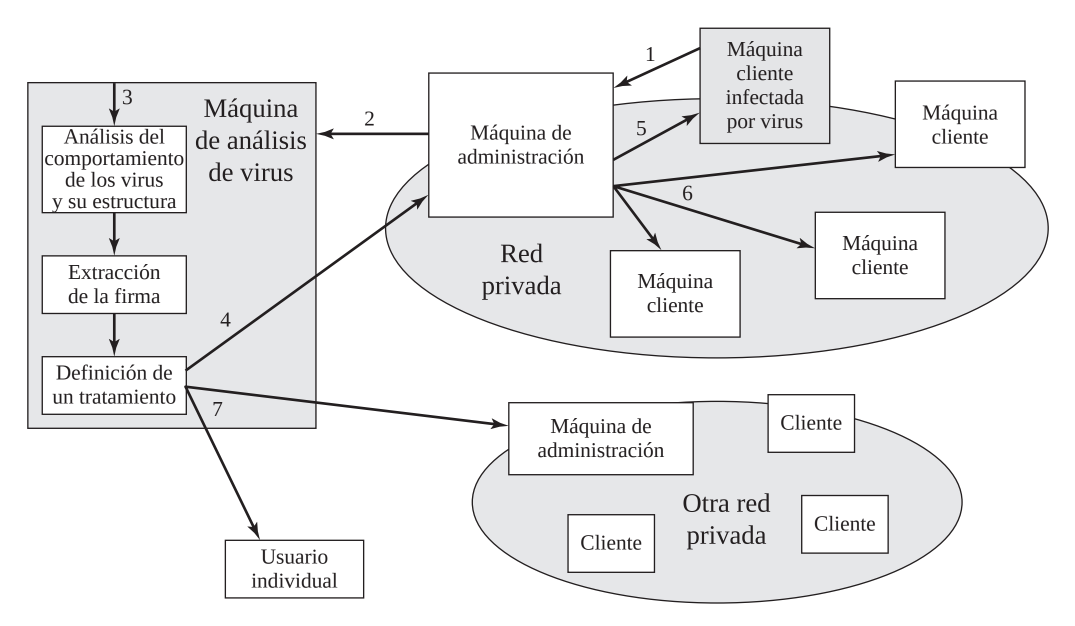


Figura 16.9. Sistema de inmunidad digital.

1. El virus se envía a sí mismo a todo el mundo en el listín de direcciones del software de correo del usuario.
2. El virus realiza daño localmente.

A finales de 1999, aparecieron unas nuevas versiones de virus por correo más potentes. Estas nuevas versiones se activaban simplemente abriendo un correo que contenía el virus en lugar de necesitar que se abriese el fichero adjunto. Los virus utilizaban el lenguaje Visual Basic soportado por el software de correo electrónico.

De esta forma vemos la aparición de una nueva generación de *malware* que llega vía correo electrónico y que utiliza las funcionalidades del software que lo maneja para replicarse a sí mismo a través de Internet. El virus se propaga tan pronto como se encuentra activo (bien abriendo el fichero adjunto o directamente el correo) a todas las direcciones conocidas por la máquina infectada. Como resultado, mientras que los virus antes solían requerir meses o años para propagarse, en la actualidad pueden hacerlo en cuestión de horas. Esto hace muy difícil a los programas antivirus responder antes de que se realice el posible daño. Finalmente, se debe situar un grado importante de seguridad en el software de utilidad y en las aplicaciones para Internet que los equipos PC tienen para poder contrarrestar el crecimiento de estas amenazas [SCHN99].

16.5. SISTEMAS CONFIABLES

Mucho de lo que hemos comentado hasta ahora se refiere a la protección de un mensaje o un dato de los ataques pasivos o activos realizados por un determinado usuario. Un requisito, hasta cierto punto diferente, pero ampliamente aplicable es la protección de datos o de recursos por medio de niveles de seguridad. Esto se encuentra habitualmente clasificado por medio de la nomenclatura militar, en la cual la información se denota como no clasificada (N), confidencial (C), secreta (S), de alto secreto (AS), o incluso niveles superiores. Este concepto es igualmente aplicable a otras áreas,

donde la información se puede organizar en grandes categorías y en las que a los usuarios se les pueden otorgar credenciales de acceso a ciertas categorías de datos. Por ejemplo, el nivel más alto de seguridad puede ser para los documentos y datos de planificación estratégica de la organización, accesibles únicamente para los directivos de la misma y su personal directo; a continuación pueden venir datos financieros sensibles y del personal, accesibles sólo por parte del personal administrativo, directivos, etc.

Cuando se definen múltiples niveles o categorías de datos, a los requisitos se les denomina **seguridad multinivel**. La directriz general para los requisitos de seguridad multinivel es que el sujeto a un nivel alto no puede compartir información con un sujeto a un nivel inferior o no comparable a menos que el flujo de información refleje la voluntad de un usuario autorizado. Por propósitos de implementación, este requisito se articula en dos partes. Un sistema seguro multinivel debe proporcionar lo siguiente:

- **No leer hacia arriba (*no read up*).** Un sujeto sólo puede leer un objeto de un nivel de seguridad igual o inferior. Esto se denominan en la literatura como **propiedad de seguridad simple**.
- **No escribir hacia abajo (*no write down*).** Un sujeto sólo puede escribir en un objeto de un nivel de seguridad igual o superior. Denominado habitualmente la literatura como la propiedad-*[1](pronunciado propiedad estrella o *star-property*).

Estas dos reglas, si se articulan de forma apropiada, proporcionan seguridad multinivel. Pero en el sistema de procesamiento de datos, la estrategia que se ha tomado, y que ha sido objeto de múltiples investigaciones y desarrollos, se basa en el concepto de monitor de referencias. Dicha estrategia se encuentra representada en la figura 16.10. El monitor de referencias es un elemento de control en el hardware con el sistema operativo de un ordenador que regula el acceso de sujetos a objetos en base a unos parámetros de seguridad de dichos participantes. El monitor de referencia tiene acceso a un fichero, conocido como base de datos del núcleo de seguridad, que muestra una lista de los privilegios acceso (habilitación) de cada sujeto y los atributos de protección (nivel de clasificación) de cada objeto. El monitor de referencia aplica las reglas de seguridad (no leer hacia arriba, no escribir hacia abajo) y tiene las siguientes propiedades:

- **Mediación completa.** Las reglas de seguridad se aplican a cada acceso, no sólo por ejemplo, a la apertura de un fichero.
- **Aislamiento.** El monitor de referencia y la base de datos están protegidos de cualquier modificación no autorizada.
- **Verificabilidad.** La corrección del monitor de referencia debe estar probada. Esto es, debe ser posible demostrar matemáticamente que el monitor de referencia aplica las reglas de seguridad y proporciona mediación completa y aislamiento.

Existen algunos de estos requisitos que son requeridos. El requisito de la mediación completa implica que todo acceso a los datos dentro de la memoria principal y del disco y cinta deben pasar por un mediador. Las implementaciones de software puras imponen unas penalizaciones de rendimiento muy altas para poder ser prácticas; la solución debe ser al menos parcialmente proporcionada por el

---

[1] El «*» no significa nada en especial. Nadie puede pensar en un nombre apropiado para una propiedad durante la escritura del primer informe de un modelo. El asterisco era un carácter comodín introducido en el borrador de forma que un editor de texto pueda encontrarlo rápidamente y reemplazar todas sus instancias una vez que se le haya dado un nombre a dicha propiedad. No se proporcionó ningún nombre en su momento, de forma que el informe se publicó con el «*» intacto.

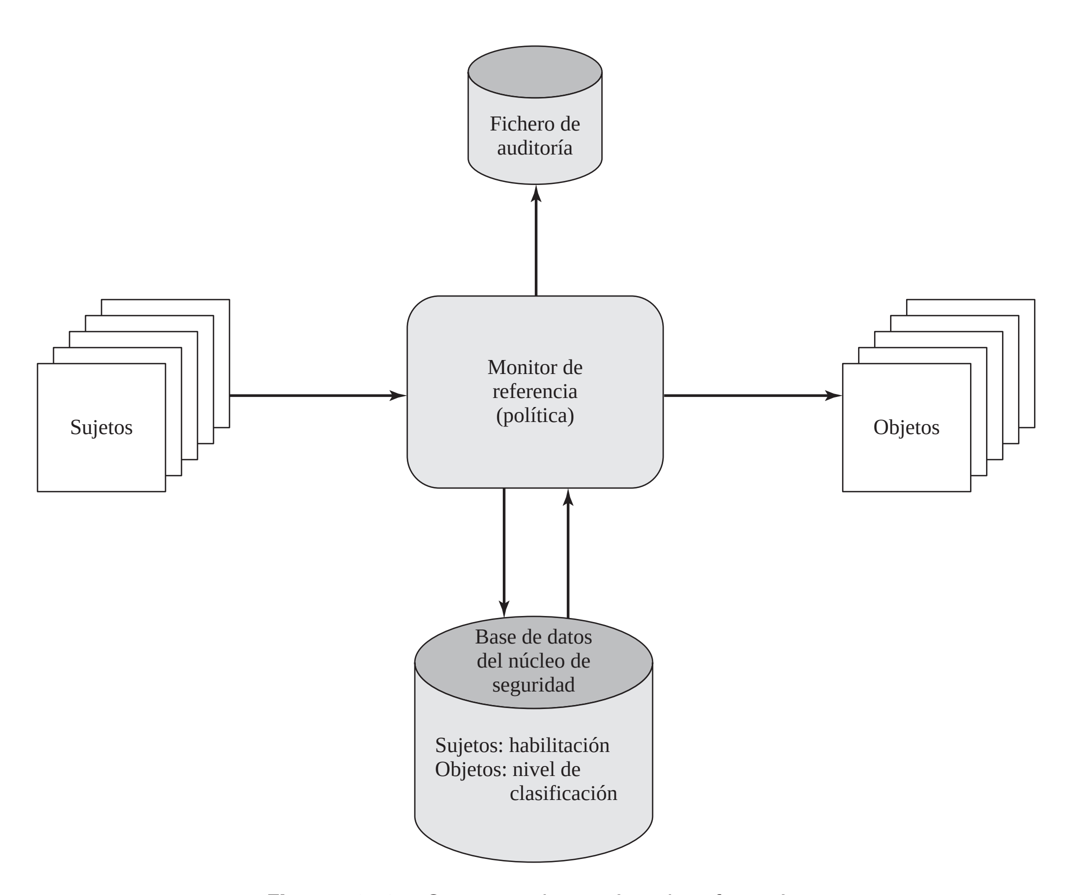


Figura 16.10. Concepto de monitor de referencia.

hardware. El requisito del aislamiento implica que no debe ser posible para un atacante, independientemente de lo listo que sea, cambiar la lógica del monitor de referencia o los contenidos de la base de datos del núcleo de seguridad. Finalmente, los requisitos para una prueba matemática pueden ser excesivos para algo tan complejo como un ordenador del propósito general. Un sistema que pueda proporcionar dicha verificación se denomina un sistema confiable.

El último elemento mostrado en la figura 16. 10 es el fichero de auditoría. El fichero de auditoría almacena eventos de seguridad importantes, por ejemplo la detección de violaciones de seguridad y los cambios autorizados en la base de datos del núcleo de seguridad.

En un esfuerzo por satisfacer sus propias necesidades y como servicio público, el Departamento de Defensa de los Estados Unidos en 1981 estableció un Centro de Seguridad de Ordenadores dentro de la Agencia de Seguridad Nacional (*National Security Agency*, NSA) con el objetivo de incentivar la difusión general de sistemas de computación confiables. Este objetivo se aplica a través del centro del Programa de Evaluación de Productos Comerciales. En esencia, este centro intenta evaluar si los productos disponibles de forma comercial cumplen los requisitos de seguridad de escritos. Este centro clasifica los productos evaluados de acuerdo a la gama de funcionalidades de seguridad que proporcionan. Estas evaluaciones se solicitan por parte de las resoluciones del Departamento de Defensa pero se encuentran publicadas y son de libre disposición. Por tanto, pueden servir como vía para clientes que deseen comprar productos comerciales y equipamientos.

DEFENSA CONTRA TROYANOS

Una vía para defenderse contra ataques realizados por medio de troyanos es el uso de sistemas operativos seguros, confiables. La Figura 16.11 muestra un ejemplo [BOEB85]. En este caso, se utiliza un troyano para evitar un mecanismo de seguridad estándar utilizado por la mayoría de los sistemas de gestión de ficheros y sistemas operativos: las listas de control de acceso. En este ejemplo, un usuario llamado Pepe interacciona por medio de un programa con un fichero de datos que contiene una cadena de caracteres críticamente sensible «CPE170KS». El usuario Pepe ha creado el fichero con permisos de lectura y escritura delegados sólo a los programas que se ejecutan por su parte: esto es, los únicos procesos que pueden acceder al fichero son aquellos cuyo propietario es Pepe.

El ataque por medio de un troyano comienza con un usuario hostil, denominado Alicia, que consigue el acceso legítimo al sistema e instala un programa troyano y un fichero privado que se utilizará durante el ataque como «bolsillo trasero» (*back pocket*). Alicia concede permisos de lectura y escritura para ella misma en este fichero y concede a Pepe permisos de sólo escritura (Figura 16.11a). Alicia induce ahora a que Pepe invoque el programa con el troyano, quizás anunciándolo como una herramienta útil. Cuando el programa detecta que Pepe es quien lo ejecuta, accede a la cadena de caracteres protegida en el fichero de dicho usuario y la copia al fichero de Alicia que antes hemos denominado bolsillo trasero (Figura 16.11b). Ambas operaciones de lectura y escritura satisfacen las restricciones impuestas por medio de las listas de control de acceso. En este momento Alicia sólo tiene que acceder al fichero que antes ha creado para acceder al valor de la cadena.

Ahora consideremos la utilización del sistema operativo seguro en este escenario (Figura 16.11c). Los niveles de seguridad se asignan a los sujetos en el momento de su acceso al sistema por medio de criterios tales como por qué terminal se ha conectado y el usuario del que se trata. Dicho usuario accede en virtud de una contraseña y de un identificador. En este ejemplo, existen dos niveles de seguridad, información sensible (gris) y pública (blanco), siendo la información sensible de un nivel mayor de seguridad que la pública. Los procesos propiedad de Pepe y su fichero de datos se consideran de nivel de seguridad sensible. El fichero y los procesos de Alicia se encuadran en el nivel de seguridad público. Si Pepe ejecuta el programa con el troyano (Figura 16.11d), el programa adquiere el nivel de seguridad de éste. De esta forma es capaz, ante la propiedad de seguridad simple, de acceder a la cadena de caracteres sensible. Cuando el programa intenta almacenar la cadena en un fichero público (el fichero que actúa como bolsillo trasero), en este caso, la propiedad-* resulta violada y el intento queda abortado por el monitor de referencia. De esta forma, el intento de escribir en un fichero, en este caso el fichero de bolsillo trasero, resulta abortado incluso aunque las listas de control de acceso lo permitan: la política de seguridad precede a los mecanismos de listas de control acceso.

16.6. SEGURIDAD EN WINDOWS

Un buen ejemplo de los conceptos de control de acceso que hemos estado revisando se encuentra en las funcionalidades de control de acceso de Windows, que aprovechan los conceptos de orientación a objetos para proporcionar unos servicios de control flexibles y potentes.

Windows proporciona unos servicios de control de acceso uniformes que se aplican a procesos, *threads*, ficheros, semáforos, ventanas y otros objetos. El control de acceso es regulado por medio de las entidades: un testigo de acceso asociado a cada proceso y un descriptor de seguridad asociado a cada objeto sobre el cuál se realizan accesos por parte de varios procesos.

ESQUEMA DE CONTROL ACCESO

Cuando el usuario se conecta al sistema Windows, el sistema operativo utiliza un esquema de nombre y contraseña para autenticar a dicho usuario. Si el acceso tiene éxito, se crea un proceso para el

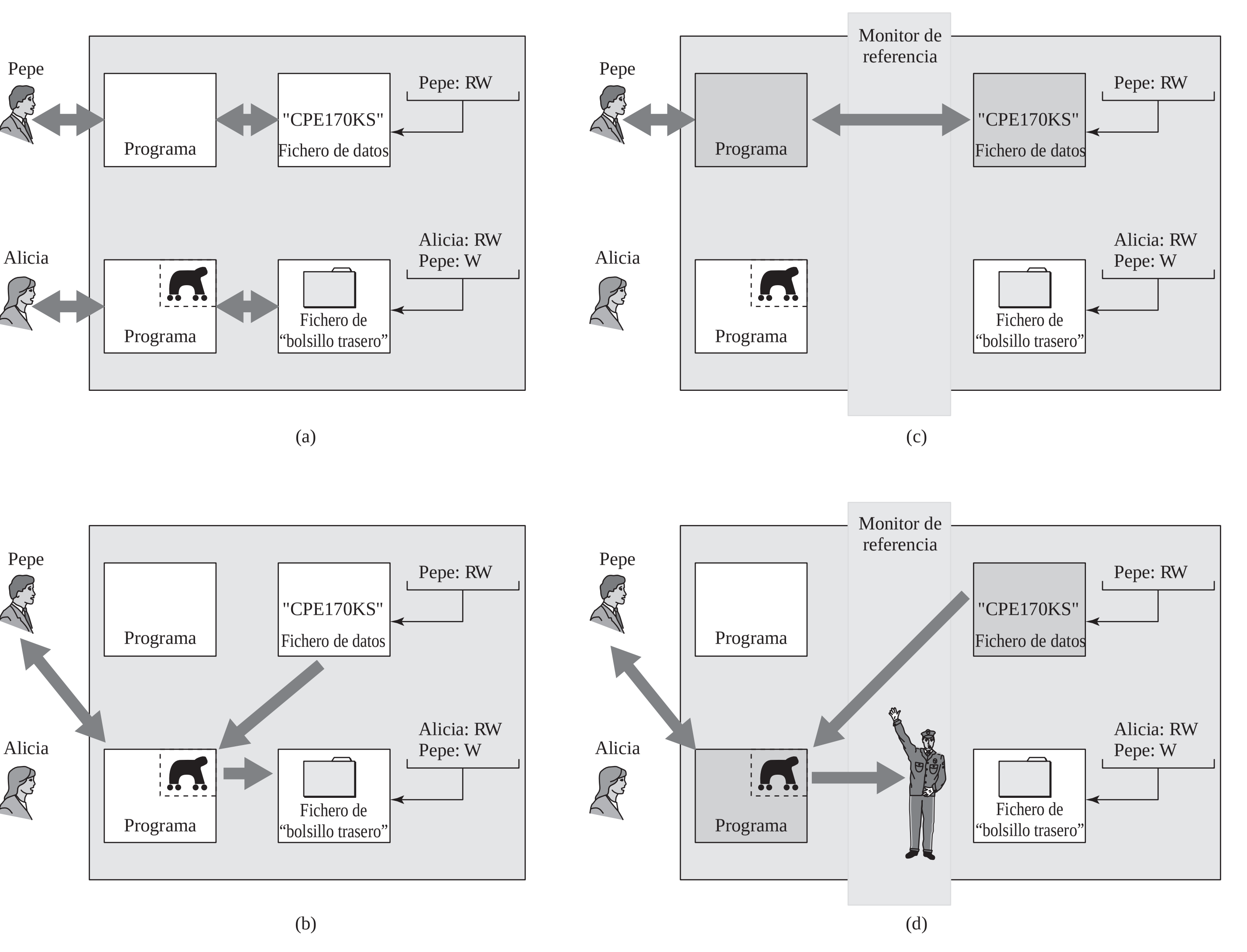


Figura 16.11. Troyanos y sistemas operativos seguros.

usuario y se le asocia un testigo de acceso al mismo. Este testigo de acceso, cuyos detalles describiremos más adelante, incluye un identificador de seguridad (SID), que es el identificador mediante el cual el usuario es conocido, a efectos de seguridad, por parte del sistema. Cuando se lanza un nuevo proceso por parte de este proceso inicial del usuario, el nuevo objeto proceso hereda el mismo testigo de acceso.

El testigo de acceso sirve para:

1. Mantener toda la información de seguridad necesaria agrupada para acelerar la validación de acceso. Cuando cualquier proceso relacionado con el usuario intenta acceder, el subsistema de seguridad puede utilizar el testigo asociado a dicho proceso para determinar los privilegios de acceso del usuario.
2. Permitir a cada proceso modificar sus características de seguridad de forma limitada sin afectar a otros procesos que se ejecuten por parte del usuario.

El significado clave del segundo punto está relacionado con los privilegios que se pueden asociar a un usuario. El testigo de acceso indica qué privilegios tiene un usuario. Generalmente, este testigo se inicializa con todos los privilegios deshabitados. A posteriori, si uno de los procesos del usuario necesita realizar una operación con privilegios, el proceso puede habilitar el privilegio apropiado e intentar el acceso. No sería deseable mantener toda la información de seguridad del usuario en un lugar común para todo el sistema, debido a que si se desea habilitar los privilegios para un proceso esto haría que se habilitasen para todos los procesos del mismo usuario.

Relacionado con cada objeto, al que es posible que varios procesos accedan, se encuentra el descriptor de seguridad. El componente clave del descriptor de seguridad es una lista de control de acceso que especifica los derechos de acceso para varios usuarios y grupos de usuarios a dicho objeto. Cuando un proceso intenta acceder a este objeto, el SID del proceso se compara con la lista de control acceso del objeto para determinar si se permite dicho acceso.

Cuando la aplicación abre una referencia a un objeto afectado por el control de seguridad, Windows verifica si el descriptor de seguridad del objeto otorga el acceso a la aplicación del usuario. Si esta verificación tiene éxito, Windows almacena los derechos de acceso que se han otorgado.

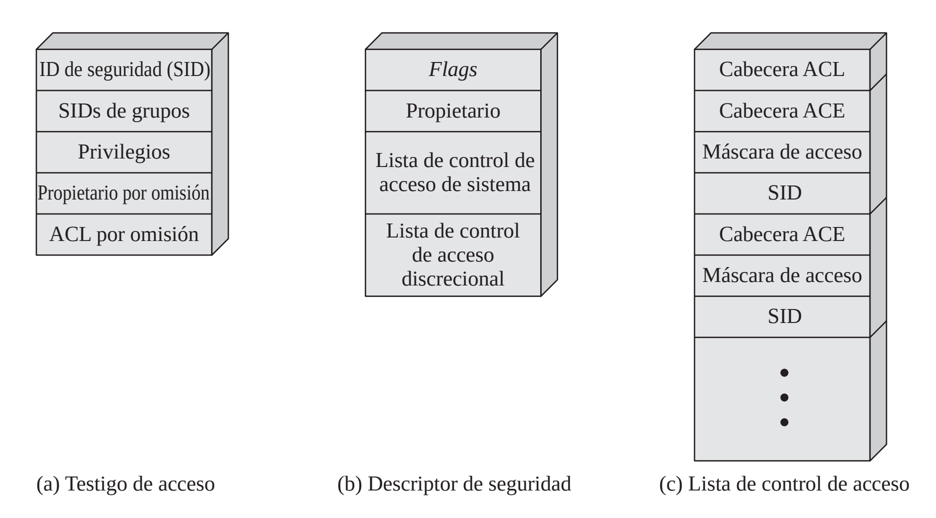


Figura 16.12. Estructuras de seguridad de Windows.

Un aspecto importante de seguridad de Windows es el concepto de personificación, que simplifica el uso de la seguridad en un entorno cliente/servidor. Si un cliente y un servidor dialogan a través de la conexión RPC, el servidor puede asumir temporalmente la identidad del cliente de forma que puede evaluar las solicitudes de acceso relativas a los permisos de dicho cliente. Después del acceso, el servidor vuelve a asumir su propia identidad.

TESTIGO DE ACCESO

La Figura 16.12a muestra la estructura general de un testigo de acceso, que incluye los siguientes parámetros:

- **Identificador de seguridad (SID).** Identifica a cada usuario de forma única en todas las máquinas de una red. Esto, habitualmente, se corresponde al nombre de usuario del sistema.
- **SIDs de grupos.** Una lista de los grupos a los cuales pertenece el usuario. Un grupo no es nada más que un conjunto de identificadores de usuario que denotan a un grupo de ellos, a efectos de control de acceso. Cada grupo tiene su SID de grupo único. El acceso a un objeto se puede definir en base a SID de grupos, SID individuales o la combinación de ambos.
- **Privilegios.** Una lista de los servicios de sistema sujetos a seguridad a los cuales este usuario puede acceder. Un ejemplo es la creación de un testigo. Otro ejemplo es dar el privilegio de *backup*; los usuarios que tienen este privilegio pueden utilizar una herramienta de *backup* para realizar copias de seguridad de los ficheros que normalmente no podrían leer. La mayoría de los usuarios no tienen privilegios adicionales.
- **Propietario por omisión.** Si este proceso crea otro objeto, este campo especifica quién es el propietario del nuevo objeto. Generalmente, el propietario de un nuevo proceso es el mismo que el propietario del proceso que lo ha lanzado. Sin embargo, un usuario puede especificar que el propietario por omisión de cualquier proceso que lance éste, en particular sea de un SID del grupo al cual él pertenece.
- **ACL por omisión.** Es la lista inicial de protecciones que se aplica a los objetos que este usuario crea. El usuario puede posteriormente alterar esta lista de control acceso para cualquier objeto que le pertenece a él o uno de sus grupos.

DESCRIPTORES DE SEGURIDAD

La Figura 16.12b muestra la estructura general de un descriptor de seguridad, que incluye los siguientes parámetros:

- ***Flags*.** Define el tipo y contenidos de un descriptor de seguridad. Los *flags* indican si están presentes o no las SACL y las DACL, si se asignan a un objeto por medio de un mecanismo por efecto o no, y si los punteros del descriptor utilizan direccionamiento absoluto o relativo. Los descriptores relativos son necesarios para los objetos que se transmiten a través de una red, por ejemplo la información transmitida en una RPC.
- **Propietario.** El propietario de un objeto puede realizar habitualmente cualquier acción sobre el descriptor de seguridad. Este propietario puede ser un SID individual o de grupo. El propietario está autorizado para cambiar los contenidos de la DACL.
- **Lista de control de acceso de sistema (*System access control list*, SACL).** Indica qué tipo de operaciones sobre dicho objeto generan mensajes de auditoría. La aplicación debe tener los pri-

vilegios correspondientes en su testigo de acceso para leer y escribir la SACL de cualquier objeto. Esto se hace para evitar que aplicaciones no autorizadas lean las SACL (conociendo por tanto lo que no deben hacer para evitar que se generen mensajes de auditoría) o escribirlas (para generar muchas trazas de auditoría que harán que una operación ilícita pase desapercibida).

- **Lista de control de acceso discrecional (*Discretionary access control list*, DACL).** Indican qué usuarios y grupos pueden acceder a este objeto y para qué operaciones. Consiste en una lista de entradas de control de acceso (ACE).

Cuando se crea un objeto, el proceso que lo crea puede asignar como propietario a su propio SID o cualquier SID de grupo que se encuentre en su testigo de acceso. El proceso que lo crea no puede asignar como propietario a un identificador que no se encuentre en su testigo de acceso. Posteriormente, cualquier proceso al que se haya concedido el derecho para cambiar el propietario de un objeto puede hacerlo, pero aplicando de nuevo la misma restricción. El motivo de esta restricción es evitar que un usuario pueda encubrir su rastro después de intentar alguna acción no autorizada.

Veamos con más detalle la estructura de las listas de control de acceso, debido a que éstas se encuentran en el corazón del servicio de control de acceso de Windows (Figura 16.12c). Cada lista consiste en una cabecera general y en un número variable de entradas de control de acceso. Cada entrada especifica un SID individual o de grupo y una máscara de acceso que define los derechos que serán otorgados a dicho SID. Cuando un proceso intenta acceder a un objeto, el manejador de objetos del ejecutivo de Windows lee el SID y los SID de grupos del testigo de acceso y recorre la DACL del objeto. Si se encuentra una equivalencia, esto es que haya una ACE que su SID coincide con el SID que se consulta, entonces a dicho proceso se le otorgan los derechos de acceso indicados en la mascara de acceso de la ACE.

La Figura 16.13 muestra los contenidos de una mascara de acceso. Los 16 bits menos significativos especifican los derechos de acceso que se aplican a un tipo de objeto en particular. Por ejemplo, el bit 0 del objeto fichero es el acceso Lectura de Datos de Fichero (File_Read_Data) y el bit 0 del objeto evento es el acceso Consulta de Estado de Evento (Event_Query_Status).

Los 16 bits más significativos de la mascara contienen los bits que se aplican a todos los tipos objetos. Cinco de éstos se denominan tipos de acceso estándar:

- **Sincronizado.** Otorgar permisos para la ejecución sincronizada con algún evento asociado a este objeto. En particular, este objeto se puede utilizar en una función de espera.
- **Escritura del propietario.** Permite a un programa modificar el propietario del objeto. Esto es útil debido a que el propietario del objeto siempre puede cambiarle su protección (al propietario no se puede negar el acceso de escritura DAC).
- **Escritura DAC.** Permite a una aplicación modificar la DACL y por tanto la protección del objeto.
- **Lectura de control.** Permite a una aplicación consultar los campos de propietario y DACL del descriptor de seguridad del objeto.
- **Borrado.** Permite a la aplicación borrar este objeto.

La mitad del orden superior de la máscara de acceso también contiene cuatro tipos de accesos genéricos. Estos bits proporcionan una forma apropiada para fijar los tipos de accesos específicos en un gran número de clases de objetos diferentes. Por ejemplo, supongamos que una aplicación desea crear distintos tipos de objetos y asegurarse que los usuarios tienen acceso de lectura a dichos objetos, incluso si la lectura tiene un significado diferente para cada uno de estos tipos de objetos. Para proteger a cada uno de los objetos de estos tipos sin unos bits de accesos genéricos, la aplicación debería cons-

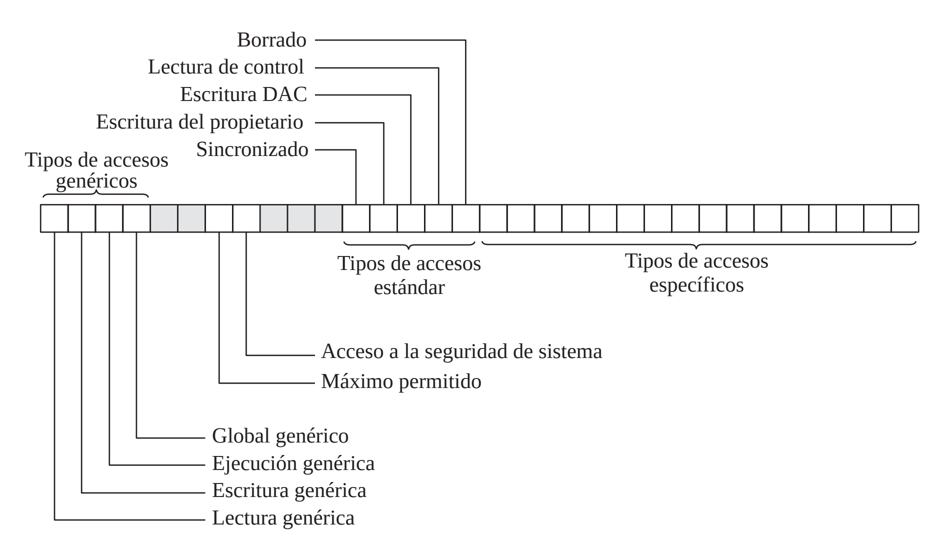


Figura 16.13. Máscara de acceso.

truir una ACE diferente para cada tipo de objeto y debería tener cuidado en pasar la ACE correcta cuando se crea cada uno de ellos. Es mucho más conveniente crear una ACE única que exprese el concepto genérico de permitir la lectura, simplemente aplicando esta ACE a cada uno de los objetos que se crean, así se resolvería el problema. Éste es el propósito de los bits de accesos genéricos, que son:

- Global genérico: permitir todos los accesos.
- Ejecución genérica: permite la ejecución si es ejecutable.
- Escritura genérica: permitir el acceso para escritura.
- Lectura genérica: permite el acceso en modo sólo lectura.

Los bits genéricos también afectan a los tipos de acceso estándar. Por ejemplo, en un objeto fichero, el bit de lectura genérica se traduce en los bits estándar de Control de Lectura y Sincronizado en los bits específicos de objeto de Lectura de Datos de Fichero (File_Read_Data), Lectura de Atributos de Fichero (File_Read_Attributes), y Lectura de EA de Fichero (File_Read_EA). Insertando una ACE en un objeto de fichero que otorgue a un SID el permiso de lectura genérica se proporcionan estos cinco derechos de acceso de la misma forma que si hubiesen sido especificados de forma individual en la máscara de acceso.

Los dos bits restantes en la máscara de acceso tienen un significado especial. El bit de Acceso a la Seguridad del Sistema permite modificar los controles de auditoría y alarmas para este objeto. Sin embargo, no es suficiente con poner este bit a 1 para un SID determinado, también hace falta que el testigo de acceso para el proceso que tenga ese SID incluya el privilegio correspondiente.

Finalmente, el bit de Máximo Permitido no es verdaderamente un bit de acceso, se trata de un modificador del algoritmo de búsqueda que utiliza Windows para la localización de un SID en una DACL. Habitualmente, Windows recorrerá la DACL hasta que encuentre una ACE que específicamente otorgue (tenga el bit a 1) o revoque (tenga el bit a 0) la solicitud de acceso para el proceso que la demanda o bien hasta que encuentre el final de la DACL, en cuyo caso el acceso se deniega. El bit de Máximo Permitido permite al propietario de un objeto definir un conjunto de

derechos de acceso que son el máximo que se pueden permitir para un determinado usuario. Con esto en mente, supongamos que una aplicación no sabe todas las operaciones que va a solicitar que se realicen sobre un objeto durante una determinada sesión. Existen tres opciones para solicitar el acceso:

1. Intentar abrir el objeto para todos los accesos posibles. La desventaja de esta estrategia es que el acceso se puede negar incluso a pesar de que la aplicación tenga todo los derechos de acceso verdaderamente necesarios para esa sesión.
2. Sólo abrir el objeto cuando el acceso específico se va a solicitar, y abrir un nuevo manejador de objeto con cada uno de los diferentes tipos de solicitudes. Éste puede ser el método habitualmente preferido debido a que no causará la denegación de accesos no necesarios, y tampoco proporcionará más accesos que los que se requieren. Sin embargo, proporciona una sobrecarga adicional.
3. Intentar abrir el objeto para el mayor número de posibles accesos que el objeto permita para este SID. La ventaja es que al usuario no se le va a denegar de forma artificial ningún acceso, pero eso sí puede darse que la aplicación tenga mayor nivel de acceso que el que necesita. Esta última situación puede ocultar fallos y errores en la aplicación.

Una característica importante de la seguridad en Windows es que las aplicaciones pueden utilizar el marco de seguridad que proporciona el sistema operativo para los objetos definidos por el usuario. Por ejemplo, un servidor de bases de datos puede crear sus propios descriptores de seguridad y asociarlos a diferentes partes de la base de datos. Adicionalmente a las restricciones de acceso de lectura/escritura, el servidor puede asegurar operaciones específicas de la base datos, tales como avanzar dentro de un conjunto de resultados o realizar operación de *join*. Será responsabilidad del servidor definir el significado de estos derechos especiales y revisar las verificaciones de acceso. Pero estas verificaciones se realizarían en un contexto estándar, utilizando cuentas de usuarios y grupos y los ficheros de auditoría que son válidos para todo el sistema. El modelo de seguridad extensible debe resultar útil para implementar sistemas de ficheros externos.

16.7. RESUMEN

Los requisitos de seguridad se encuentran mejor garantizados examinando las diferentes amenazas de seguridad a las que se enfrenta una organización. La interrupción del servicio es una amenaza contra la disponibilidad. La intercepción de información es una amenaza contra la privacidad. Y finalmente, tanto la modificación de información íntima como la fabricación de información de forma no autorizada son amenazas para la integridad.

Un área clave de la seguridad informática es la protección de memoria. Esto es esencial en un sistema en el cual múltiples procesos se encuentran activos en el mismo instante. Los esquemas de memoria virtual están capacitados típicamente con los mecanismos apropiados para realizar esta tarea.

Otra técnica de seguridad importante es el control de acceso. El control de acceso debe asegurar que sólo los usuarios autorizados tienen acceso a un sistema en particular y a sus recursos individuales y el acceso y la modificación de partes de estos datos se encuentra limitado a individuos y programas que estén autorizados. Hablando de forma estricta, el control de acceso es una cuestión de seguridad informática más que de seguridad de red. Esto es, en la mayoría de casos, un mecanismo de control de acceso se implementa dentro de un único ordenador para controlar el acceso a dicho equipo. Sin embargo, debido a que muchos de los accesos a un ordenador se realizan por medio de redes o servicios de comunicaciones, los mecanismos de control de acceso deben diseñarse para operar de forma efectiva en un entorno distribuido de red.

Una amenaza creciente es la causada por los virus y mecanismos software similares. Estas amenazas aprovechan las vulnerabilidades del software del sistema bien para ganar acceso de forma no autorizada a la información o para degradar el servicio del sistema.

Una tecnología que está permitiendo aumentar la seguridad de los sistemas y aplicaciones de entornos comerciales y militares son los sistemas confiables. Un sistema confiable proporciona los mecanismos para regular el acceso a los datos basándose en quién se encuentra autorizado para acceder a qué cosa. El punto clave se encuentra en que el sistema está diseñado e implementado de forma que los usuarios tienen la seguridad completa de que el sistema va a forzar a que se siga una política de seguridad determinada.

16.8. LECTURAS RECOMENDADAS Y SITIOS WEB

Los contenidos de este capítulo se encuentran tratados con más detalle en [STAL03]. Para cubrir la parte de algoritmos criptográficos, [SCHN96] es un trabajo de referencia esencial; contiene la descripción de numerosos algoritmos y protocolos criptográficos. Se puede encontrar una buena revisión de los aspectos relativos a sistemas operativos en [PIEP03], [GOLL99], y [PFLE97]. Por su parte, [KENT00] y [MCHU99] son dos buenos artículos recapitulatorios sobre detección de intrusos. Para una visión más profunda sobe los virus, [HARL01] es el libro a leer. [CASS01], [FORR97], [KEPH97], y [NACH97] son unos buenos artículos de revisión sobre virus y gusanos.

[COX00] y [CLER04] proporcionan una visión detallada sobre la seguridad en Windows; dirigidos tanto a administración como gestión, ambos muestran diversos aspectos internos de Windows concernientes a la seguridad.

CASS01 Cass, S. «Anatomy of Malice.» IEEE Spectrum, Noviembre 2001.

CLER04 Clercq, J. *Windows Server 2003 Security Infrastructure: Core Security Features*. Burlington, MA: Digital Press, 2004.

COX00 Cox, P., and Sheldon, T. *Windows NT Security Handbook*. New York: Osborne McGraw-Hill, 2000.

FORR97 Forrest, S.; Hofmeyr, S.; and Somayaji, A. «Computer Immunology.» Communications of the ACM, Octubre 1997.

GOLL99 Gollmann, D. *Computer Security*. New York: Wiley, 1999.

HARL01 Harley, D.; Slade, R.; and Gattiker, U. *Viruses Revealed*. New York: Osborne/McGraw-Hill, 2001.

KENT00 Kent, S. «On the Trail of Intrusions into Information Systems.» *IEEE Spectrum*, Diciembre 2000.

KEPH97 Kephart, J.; Sorkin, G.; Chess, D.; and White, S. «Fighting Computer Viruses.» *Scientific American*, Noviembre 1997.

MCHU00 McHugh, J.; Christie, A.; and Allen, J. «The Role of Intrusion Detection Systems.» IEEE Software, Septiembre/Octubre 2000.

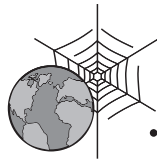
NACH97 Nachenberg, C. «Computer Virus-Antivirus Coevolution.» Communications of the ACM, Enero 1997.

PEIP03 Pieprzyk, J.; Hardjono, T.; and Seberry, J. *Fundamentals of Computer Security*. New York: Springer, 2003.

PFLE97 Pfleeger, C. *Security in Computing*. Upper Saddle River, NJ: Prentice Hall PTR, 1997.

SCHN96 Schneier, B. *Applied Cryptography*. New York: Wiley, 1996.

STAL03 Stallings, W. *Cryptography and Network Security: Principles and Practice*, 3rd edition. Upper Saddle River, NJ: Prentice Hall, 2003.



PÁGINAS WEB RECOMENDADAS:

- **Computer Security Resource Center.** Mantenido por el Instituto Nacional de Estándares y Tecnología (NIST). Contiene una amplia gama de información sobre amenazas de seguridad, tecnología y estándares.
- **CERT Coordination Center.** Una organización que creció bajo el amparo del equipo de respuesta de emergencia informática creado por la Agencia de Proyectos de Investigación Avanzada para la Defensa (DARPA). Contiene buena información relativa a las amenazas de Seguridad vía Internet, vulnerabilidades, y estadísticas de ataque.
- **Intrusión Detection Working Group.** Incluye todos los documentos generados por dicho grupo.
- **AntiVirus On-line.** Página web de IBM sobre información de virus; una de las mejores.
- **Vmyth.** Dedicado a mostrar falsos virus y a esclarecer errores de concepto sobre los virus reales.

16.9. TÉRMINOS CLAVE, CUESTIONES DE REPASO Y PROBLEMAS

TÉRMINOS CLAVE

amenaza activa	control de acceso	RSA
amenaza pasiva	detección de intrusos	respuesta
Autenticidad	denegación de servicio	sistema confiable
bomba lógica	disponibilidad	software malicioso (malware)
cifrado	estándar de cifrado avanzado (AES)	triple DES (3DES)
cifrado convencional	estándar de cifrado de datos (DES)	troyano
cifrado de clave pública	gusano	virus
cifrado simétrico	integridad	virus basados en macros
confidencialidad	intruso	virus de correo
contraseña	puerta secreta	zombie

CUESTIONES DE REPASO

16.1. ¿Cuáles son los principales requisitos tratados por la seguridad informática?

16.2. ¿Cuál es la diferencia entre las amenazas de seguridad activas y pasivas?

16.3. Proporcione una lista y una breve descripción de las diferentes categorías de amenazas de seguridad activas y pasivas.

16.4. ¿Qué elementos se necesitan para la técnica de control de acceso más habitual?

16.5. En el control de acceso, ¿cuál es la diferencia entre sujeto y objeto?

16.6. Explique el propósito del aderezo en la Figura 16.6.

16.7. Explique las diferencias entre la detección de intrusos por medio de la estadística de anomalías y la basada en reglas.

16.8. El desarrollo de contenidos adjuntos y *malware* VBS de correo en 1999 y 2000 (por ejemplo, Melisa y *love letter*) se denominan en la prensa virus de correo. ¿Sería más apropiado el término de gusanos de correo?

16.9. ¿Cuál es el papel que tuvo el cifrado en el desarrollo de los virus?

16.10. ¿Cuáles son las dos estrategias típicas para atacar un esquema de cifrado típico?

16.11. ¿Qué son DES y triple DES?

16.12. ¿Cómo se espera que AES mejore a triple DES?

16.13. ¿Qué criterio de evaluación se usa para asignar los candidatos en AES?

16.14. Explique las diferencias entre el cifrado convencional y el de clave pública.

16.15. ¿Cuál es la distinción entre los términos de clave pública, clave privada y clave secreta?

PROBLEMAS

16.1. Asuma que las contraseñas se seleccionan entre las combinaciones de cuatro caracteres de los 26 caracteres del alfabeto. Suponga que un adversario es capaz de intentar contraseñas a una tasa de una por segundo.

a) Suponiendo que no se tiene ninguna respuesta hacia el adversario hasta que se ha realizado cada intento, ¿cuánto tiempo se espera que se tardará en descubrir la contraseña correcta?

b) Suponiendo que sí se tiene respuesta por medio de un indicador que marque cada uno de los caracteres erróneos de la contraseña, ¿cuánto tiempo se espera que tardará en descubrir la contraseña correcta?

16.2. Asuma que un elemento origen de longitud k se proyecta de una forma uniforme sobre un elemento destino de longitud p. Si cada uno de los dígitos puede tomar uno de los r valores posibles, siendo, por tanto el número de elementos fuente r^k y el número de elementos destino es un número menor i. Un elemento origen en particular x_i se proyecta en un elemento destino en particular y.

a) ¿Cuál es la probabilidad de que un adversario pueda seleccionar el elemento original correcto en un solo intento?

b) ¿Cuál es la probabilidad de que dos elementos originales diferentes x_l ($x_i \neq x_l$) que se proyecta en el mismo elemento destino, y_j, pueda producirse por parte de un adversario?

c) ¿Cuál es la probabilidad de que un adversario pueda seleccionar el elemento destino correcto en un solo intento?

16.3. Un generador de contraseñas fonético toma dos segmentos de forma aleatoria por cada contraseña de seis letras. El formato de cada uno de los segmentos es CVC (consonante, vocal, consonante), donde V=<a, e, i, o, u> y $C = \overline{V}$.

a) ¿Cuál es la población completa de contraseñas?

b) ¿Cuál es la probabilidad de que un adversario adivine la contraseña correcta?

16.4. Suponga que las contraseñas están limitadas a usar los 95 caracteres imprimibles del código ASCII y que todas las contraseñas tienen una longitud de 10 caracteres. Suponiendo un *password cracker* que tiene una tasa de cifrado de 6.4 millones de cifrados por segundo.

¿Cuánto tiempo llevaría examinar de forma exhaustiva todas las posibles contraseñas del sistema?

16.5. Debido al riesgo conocido del sistema de contraseñas de UNIX, la documentación de SunOS-4.0 recomienda que se elimine el fichero de contraseñas y se sustituya por un fichero de lectura público denominado /etc/publickey. Una entrada en dicho fichero para el usuario A consistirá en el identificador ID_A, la clave pública del usuario, KU_A, y la clave privada correspondiente KR_A. Esta clave privada está cifrada usando DES con una clave derivada de la contraseña del usuario P_A. Cuando A entra en el sistema se descifra $E_{PA}[KR_A]$ para obtener KR_A. Utilizamos la notación $E_x[a]$ para indicar el cifrado o descifrado por medio de la clave x.

a) El sistema verifica posteriormente si la clave P_A que se ha proporcionado es la correcta. ¿Cómo?

b) ¿Cómo un oponente puede atacar este sistema?

16.6. El sistema de cifrado para las contraseñas de UNIX es unidireccional; no es reversible. Por tanto, ¿sería correcto decir que se trata verdaderamente de una función *hash*, y no de un mecanismo de cifrado?

16.7. Se comentó que la inclusión del valor de aderezo en el esquema de contraseñas de UNIX incrementaba la dificultad de adivinar las contraseñas en un factor de 4096. Pero este valor de aderezo se encuentra almacenado en claro en la misma entrada de la correspondiente contraseña cifrada. Por tanto, estos dos caracteres son conocidos por parte del atacante y no es necesario adivinarlos. ¿Por qué se asegura que incluir el valor de aderezo incrementa la seguridad?

16.8. Suponiendo que se ha respondido correctamente al problema anterior y se ha comprendido cuál es la importancia del valor de aderezo, he aquí otra cuestión. ¿No sería posible desmantelar todos los intentos de ataque por medio de *password crackers* incrementando ostensiblemente el tamaño del valor de aderezo, pongamos en 24 o 48 bits?

16.9. Se plantea la cuestión de si es posible o no desarrollar un programa capaz de analizar un fragmento de software para determinar si es un virus. Considere, por ejemplo, que tenemos un programa D que se supone que es capaz de hacerlo. Esto es, para cualquier programa P, si ejecutamos D(P), el resultado es TRUE (si P es un virus) o FALSE (si P no es un virus). Consideremos ahora el siguiente programa:

```
Programa CV :=
    {...
    programa-principal:=
        { if(D(CV) then goto siguiente
            else infectar-ejecutable
        }
siguiente:
    }
```

En el programa anterior, infectar-ejecutable es un módulo que analiza la memoria en busca de programas ejecutables y se replica a sí mismo en dichos programas. Determine si D puede decidir correctamente si CV es un virus.

16.10. La necesidad de la regla de «no leer hacia arriba» (*no read up*) para un sistema de seguridad multinivel es obvia. ¿Cuál es la importancia de la regla de «no escribir hacia abajo» (*no write down*)?

16.11. En la Figura 16.11 un enlace de la cadena de copia-y-observación del troyano está roto. Existen otras dos posibilidades de ángulos de ataque por parte de Alicia: Alicia entrando en el sistema e intentando leer la cadena directamente, y Alicia asignando el nivel de seguridad sensible al fichero que actúa de bolsillo trasero. ¿El monitor de referencia evita estos ataques?

16.12. Suponga que alguien le sugiere el siguiente mecanismo para confirmar que ustedes dos poseen la misma clave secreta. Usted genera una cadena de bits de la longitud de la clave, realiza la operación XOR con la clave, y manda el resultado por la red. La otra parte realiza de nuevo la operación XOR al bloque recibido con su clave (que se supone que es la misma) y se lo devuelve. Usted lo verifica y ve si coincide con la cadena original que se ha generado de forma aleatoria, de esta forma usted verifica que el otro extremo tiene la misma clave secreta, sin necesidad de que ninguno de los dos haya transmitido su clave. ¿Existe algún fallo en este mecanismo?

APÉNDICE 16A CRIPTOGRAFÍA

La tecnología esencial que subyace virtualmente en todas las aplicaciones de automatización de redes y de seguridad de computadores es la criptografía. Se utilizan dos enfoques fundamentales: criptografía simétrica, también conocida como criptografía convencional, y criptografía de clave-pública, también conocida como criptografía asimétrica. Este apéndice proporciona una visión general de ambos tipos de criptografía, junto con una breve exposición de algunos algoritmos criptográficos importantes.

CRIPTOGRAFÍA SIMÉTRICA

La criptografía simétrica fue el único tipo de criptografía en uso antes de la introducción de la criptografía de clave-pública a finales de los años 70. La criptografía simétrica se ha usado para comunicaciones secretas por incontables individuos y grupos, desde Julio Cesar a la fuerza naval alemana hasta los diplomáticos, militares y comerciantes de la actualidad. Esta continúa siendo con mucho la más usada de los dos tipos de criptografía.

Un esquema de criptografía simétrica tiene cinco ingredientes (Figura 16.14):

- **Texto en claro.** Este es el mensaje o datos originales que alimentan la entrada del algoritmo.
- **Algoritmo de cifrado.** El algoritmo de cifrado realiza varias sustituciones y transformaciones sobre el texto en claro.
- **Clave secreta.** La clave secreta es también una entrada para el algoritmo de cifrado. Las sustituciones y transformaciones concretas que el algoritmo realice dependen de la clave.
- **Texto cifrado.** Este es el mensaje codificado producido como salida. Depende del texto en claro y de la clave secreta. Para un mensaje dato, dos claves diferentes producirán dos textos cifrados diferentes.
- **Algoritmo de descifrado.** Este es esencialmente el algoritmo de cifrado ejecutado al revés. Toma el texto cifrado y la clave secreta y produce el texto en claro original.

Hay dos requisitos para el uso seguro de la criptografía simétrica:

1. Necesitamos un algoritmo criptográfico sólido. Como mínimo, queremos un algoritmo tal que un oponente que conozca el algoritmo y tenga acceso a uno o más mensajes cifrados sea inca-

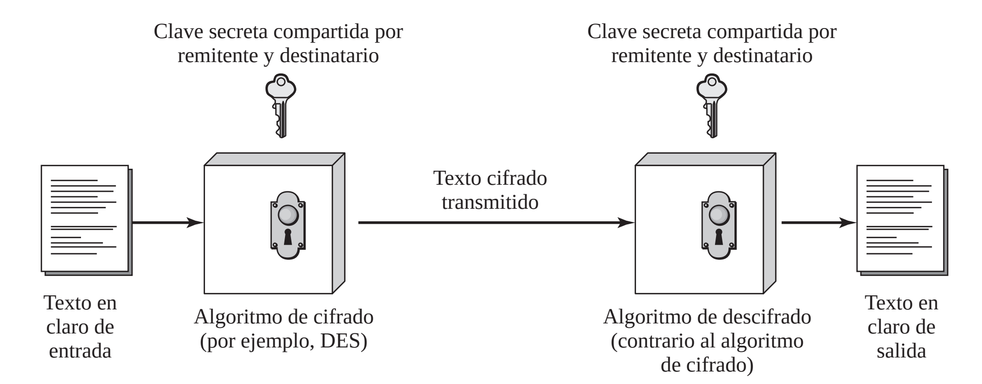


Figura 16.14. Modelo simplificado de criptografía simétrica.

paz de descifrar dichos mensajes o de deducir la clave. Normalmente este requisito se enuncia de una manera más rotunda. El oponente debe de ser incapaz de descifrar el texto cifrado o de descubrir la clave incluso si el o ella está en posesión de varios textos cifrados junto con el texto en claro que produjo dichos textos cifrados.

2. Emisor y receptor deben haber obtenido copias de la clave secreta de manera segura y deben guardar la clave con seguridad. Si alguien puede descubrir la clave y conoce el algoritmo, toda comunicación que use esta clave será legible.

Hay dos aproximaciones genéricas para atacar un esquema de criptografía simétrica. El primer ataque se conoce como **criptoanálisis**. Los ataques criptoanalíticos se basan en la naturaleza del algoritmo y quizás también en cierto conocimiento de características generales del texto en claro o incluso en algunos ejemplos de parejas texto en claro-texto cifrado. Este tipo de ataque explota las características del algoritmo para intentar deducir un texto en claro específico o deducir la clave que se está usando. Si el ataque tiene éxito en deducir la clave, el efecto es catastrófico: todos los mensajes futuros y pasados cifrados con dicha clave están comprometidos.

El segundo método, conocido como ataque por fuerza-bruta, es probar cada posible clave sobre un trozo de texto cifrado hasta que se consiga transformar en un texto en claro inteligible. De media, deben probarse la mitad de las posibles claves para conseguir el éxito. La Tabla 16.4 muestra cuánto tiempo se necesita para claves de varios tamaños. La tabla muestra resultados para cada tamaño de clave, asumiendo que un simple descifrado tarda en realizarse 1µs, un orden de magnitud razonable para los computadores de hoy día. Con el uso masivo de microprocesadores organizados en paralelo, puede ser posible alcanzar tasas de procesamiento varios órdenes de magnitud mayores. La última columna de la tabla considera los resultados de un sistema que puede procesar un millón de claves por microsegundo. Como puede verse, a este nivel de prestaciones, una clave de 56 bits no puede considerarse computacionalmente segura.

Los algoritmos de cifrado simétricos usados más comúnmente son los cifradores de bloque. Un cifrador de bloque procesa el texto en claro introducido en bloques de tamaño fijo y produce un bloque de texto cifrado de igual tamaño por cada bloque de texto en claro. Los dos algoritmos simétricos más importantes, los cuales son ambos cifradores de bloque, son el *Data Encryption Standard* (DES) y el *Advanced Encryption Standard* (AES).

El *Data Encryption Standard* (DES). El DES ha sido el algoritmo criptográfico dominante desde su introducción en 1977. Sin embargo, dado que el DES utiliza claves de sólo 56 bits, es sólo cuestión de tiempo que la velocidad de proceso de los computadores lo dejen obsoleto. En 1988, la *Elec-*

tronic Frontier Foundation (EFF) anunció que habría roto el DES usando una máquina «rompe DES» de propósito específico construida por menos de 250.000 dólares. El ataque llevó menos de tres días. La EFF ha publicado una descripción detallada de la máquina, permitiendo a otros construir la suya propia [EFF98]. Y, por supuesto, el hardware continúa abaratándose e incrementando su velocidad, haciendo el DES inútil.

La vida del DES se alargó mediante el uso del triple DES (3DES), que conlleva repetir el algoritmo DES básico tres veces, usando dos o bien tres claves distintas, para conseguir una clave de 112 o 168 bits.

La principal desventaja del 3DES es que la implementación software del algoritmo es relativamente lenta. Una segunda desventaja es que tanto el DES como el 3DES usan bloques de tamaño 64 bits. Es deseable un tamaño de bloque mayor por razones de eficiencia y seguridad.

Tabla 16.4. Tiempo Medio Necesario para la Búsqueda Exhaustiva de Clave.

Tamaño de clave (bits)	Número de claves alternativas	Tiempo necesario a 1 cifrado/μs	Tiempo necesario a 10^6 cifrado/μs
32	$2^{32} = 4.3 \times 10^9$	2^{31} μs = 35.8 minutos	2.15 milisegundos
56	$2^{56} = 7.2 \times 10^{16}$	2^{55} μs = 1142 años	10.01 horas
128	$2^{128} = 3.4 \times 10^{38}$	2^{127} μs = 5.4×10^{24} años	5.4×10^{18} años
168	$2^{168} = 3.7 \times 10^{50}$	2^{167} μs = 5.9×10^{36} años	5.9×10^{30} años
26 caracteres (permutación)	$26! = 4 \times 10^{26}$	2×10^{26} μs = 6.4×10^{12} años	6.4×10^6 años

Advanced Encryption Standard. Debido a estas desventajas, el 3DES no es realmente un candidato para su uso a largo plazo. Como sustituto, el *National Institute of Standards and Technology* (NIST) solicitó en 1977 propuestas para un algoritmo nuevo, denominado *Advanced Encryption Standard* (AES), que debería tener un nivel de seguridad igual o mejor que el 3DES y una eficiencia significativamente mejorada. Además de estos requisitos generales, el NITS especificó que el AES debe ser un cifrador simétrico de bloque con bloques de longitud 128 bits y soportar claves de longitud 128, 192 y 256 bits. Los criterios de evaluación incluyen seguridad, eficiencia computacional, necesidades de memoria, idoneidad hardware y software y flexibilidad. En 2001, NITS publicó el AES como un estándar federal de procesamiento de información (FIPS 197).

CRIPTOGRAFÍA DE CLAVE PÚBLICA

La criptografía de clave pública, públicamente propuesta por primera vez por Diffie y Hellman en 1976, es el primer avance realmente revolucionario en la criptografía en literalmente miles de años. Esto se debe a la siguiente razón, los algoritmos de clave pública se basan en funciones matemáticas en vez de en simples operaciones sobre patrones de bits. Más importante, la criptografía de clave pública es asimétrica, requiriendo el uso de dos claves separadas, en contraste con la criptografía simétrica, que utiliza solamente una clave. El uso de dos claves tiene consecuencias profundas en las áreas de confidencialidad, distribución de clave y autenticación.

Antes de seguir, primero debemos mencionar varias ideas comúnmente equivocadas acerca de la criptografía de clave pública. Una es que la criptografía de clave pública es más segura al criptoanálisis que la criptografía simétrica. De hecho, la seguridad de un esquema criptográfico depende de la longitud de la clave y del trabajo computacional necesario para romper el cifrado. No hay en principio

nada en la criptografía simétrica o la de clave pública que haga a una superior a la otra desde el punto de vista de resistir al criptoanálisis. Una segunda idea equivocada es que la criptografía de clave pública es una técnica de propósito general y que ha dejado obsoleta a la criptografía simétrica. Por el contrario, dadas las necesidades computacionales de los esquemas criptográficos de clave pública actuales, no parece previsible que se abandone la criptografía simétrica. Finalmente, hay la creencia de que cuando se utiliza criptografía de clave pública la distribución de la clave es trivial, comparado con el complicado protocolo para la criptografía simétrica que involucra centros de distribución de claves. De hecho, se necesita cierto tipo de protocolo, involucrando a menudo a un agente central, y los procedimientos necesarios no son más simples ni más eficientes que los requeridos para criptografía simétrica.

Un esquema criptográfico de clave pública tiene seis ingredientes (Figura 16.15):

- **Texto en claro.** Este es el mensaje o datos originales que alimentan la entrada del algoritmo.
- **Algoritmo de cifrado.** El algoritmo de cifrado realiza varias sustituciones y transformaciones sobre el texto en claro.
- **Claves pública y privada.** Este es un par de claves que han sido seleccionadas de manera que si una se utiliza para cifrar, la otra se usa para descifrar. Las transformaciones concretas que realiza el algoritmo de cifrado dependen de la clave pública o privada que se proporciona como entrada.
- **Texto cifrado.** Este es el mensaje codificado producido como salida. Depende del texto en claro y de la clave secreta. Para un mensaje dato, dos claves diferentes producirán dos textos cifrados diferentes.
- **Algoritmo de descifrado.** Este algoritmo acepta el texto cifrado y la correspondiente clave y produce el texto en claro original.

El proceso funciona (produce como salida el texto en claro correcto) sin importar el orden en que se utiliza la pareja de claves. Como el nombre sugiere, la clave pública del par se hace pública para que la usen otros, mientras que la clave privada es solamente para su propietario.

Ahora, digamos que Pepe quiere enviar un mensaje privado a Alicia y supongamos que él tiene la clave pública de Alicia y Alicia tiene la correspondiente clave privada (Figura 16.15a). Usando la clave pública de Alicia, Pepe cifra el mensaje para producir texto cifrado. Entonces el texto cifrado se lo transmite a Alicia. Cuando Alicia recibe el texto cifrado, lo descifra usando su clave privada. Dado que sólo Alicia tiene una copia de su clave privada, nadie más puede descifrar el mensaje.

La criptografía de clave pública puede usarse de otro modo, tal como ilustra la Figura 16.15b. Supongamos que Pepe quiere enviar un mensaje a Alicia y, aunque no es importante que el mensaje permanezca secreto, él quiere que Alicia esté segura de que el mensaje es efectivamente de él. En este caso Pepe usa su propia clave privada para cifrar el mensaje. Cuando Alicia recibe el texto cifrado, descubre que puede descifrarlo con la clave pública de Pepe, probándose así que el mensaje debe haber sido cifrado por Pepe: nadie más tiene la clave privada de Pepe y por tanto nadie más puede haber creado un texto cifrado que puede ser descifrado con la clave pública de Pepe.

Un algoritmo criptográfico de clave pública de propósito general se basa en una clave para cifrar y una clave diferente pero relacionada para descifrar. Más aún, estos algoritmos tienen las siguientes importantes características:

- Es computacionalmente irrealizable determinar la clave de descifrado conociendo solamente el algoritmo criptográfico y la clave de cifrado.
- Cualquiera de las dos claves relacionadas puede usarse para el cifrado, y usarse la otra para el descifrado.

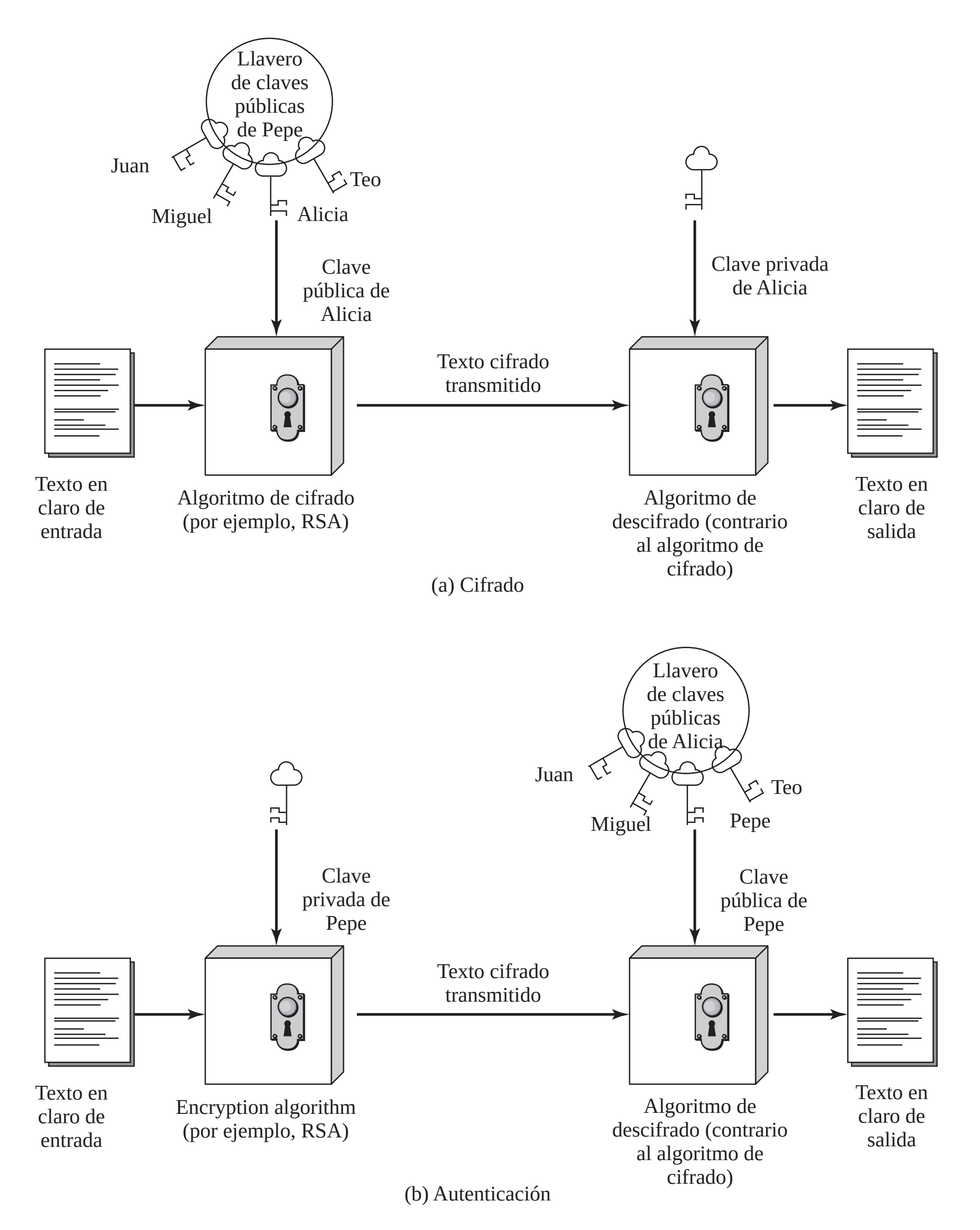


Figura 16.15. Criptografía de clave pública.

Los pasos esenciales son los siguientes:

1. Cada usuario genera un par de claves para ser usadas en el cifrado y descifrado de mensajes.
2. Cada usuario pone una de las dos claves en un registro público u otro fichero accesible. Esta es la clave pública. La clave acompañante se conserva privada. Como sugiere la Figura 16.15a, cada usuario mantiene una colección de claves públicas obtenidas de otros usuarios.
3. Si Pepe desea enviar un mensaje privado a Alicia, Pepe cifra el mensaje usando la clave pública de Alicia.

4. Cuando Alicia recibe un mensaje, lo descifra usando su clave privada. Ningún otro receptor puede descifrar el mensaje porque sólo Alicia conoce su propia clave privada.

Con este sistema, todos los participantes tienen acceso a claves públicas, y las claves privadas son generadas localmente por cada participante y por tanto nunca necesitan ser distribuidas. En la medida en que el usuario proteja su clave privada, la comunicación entrante es segura. En cualquier momento, un usuario puede cambiar su clave privada y publicar la clave pública correspondiente para reemplazar su antigua clave pública.

La clave utilizada en criptografía simétrica se conoce normalmente como **clave secreta**. Las dos claves utilizadas para criptografía de clave pública se conocen como **clave pública** y **clave privada**. Invariablemente, la clave privada se mantiene secreta, pero nos referimos a ella como clave privada en vez de clave secreta para evitar confusión con la criptografía simétrica.

Algoritmo Rivest-Shamir-Adleman (RSA). Uno de los primeros esquemas de clave pública fue desarrollado en 1977 por Ron Rivest, Adi Shamir y Len Adleman en el MIT. El esquema RSA ha reinado desde aquel momento como el único enfoque ampliamente aceptado e implementado de criptografía de clave pública. El RSA es un cifrador en el cual el texto en claro y el texto cifrado son enteros entren 0 y $n - 1$ para algún n. El cifrado requiere aritmética modular. La fortaleza del algoritmo se basa en la dificultad de factorizar números en sus factores primos.

APÉNDICE A

Temas de concurrencia

A1. EXCLUSIÓN MUTUA. TÉCNICAS DE SOFTWARE

Se pueden implementar diferentes técnicas de software para procesos concurrentes que se ejecutan en un único procesador o una máquina multiprocesador con memoria principal compartida. Estas técnicas normalmente asumen exclusión mutua elemental a nivel de acceso a memoria ([LAMP91], *véase* el Problema A.3). Es decir, se serializan accesos simultáneos (lectura y/o escritura) a la misma ubicación de memoria principal por alguna clase de árbitro de memoria, aunque no se especifique por anticipado el orden de acceso resultante. Más allá de esto, no se asume soporte de hardware, sistema operativo o lenguaje de programación.

ALGORITMO DE DEKKER

Dijkstra [DIJK65] describió un algoritmo de exclusión mutua para dos procesos, diseñado por el matemático holandés Dekker. Siguiendo a Dijkstra, vamos a desarrollar la solución paso a paso. Esta técnica tiene la ventaja de mostrar muchos de los errores típicos del desarrollo de programas concurrentes.

Primera tentativa

Como se mencionó anteriormente, cualquier aproximación a la exclusión mutua debe basarse en algunos mecanismos fundamentales de exclusión en el hardware. El mecanismo más común es la restricción de que en un determinado momento sólo se pueda hacer un acceso a una ubicación de memoria. Utilizando esta restricción, se reserva una ubicación de memoria global etiquetada como `turno`. Un proceso (P0 ó P1) que desee ejecutar su sección crítica examina primero el contenido de `turno`. Si el valor de `turno` es igual al número del proceso, entonces el proceso puede acceder a su sección crítica. En caso contrario, está forzado a esperar. El proceso que espera lee de forma repetida el valor de la variable turno hasta que puede entrar en la sección crítica. Este procedimiento se conoce como **espera activa**, porque el proceso que no logra entrar en la sección crítica no hace nada productivo hasta que obtiene los permisos para entrar en su sección crítica. Por el contrario, debe permanecer comprobando periódicamente la variable; esto consume tiempo de procesador (espera activa) mientras espera su oportunidad.

Después de que un proceso ha obtenido el acceso a su sección crítica y después de que ha completado dicha sección, debe actualizar el valor de turno para el resto de procesos.

En términos formales, hay una variable global compartida:

int turno = 0;

La Figura A.1a muestra el programa para dos procesos. Esta solución garantiza la propiedad de exclusión mutua, pero tiene dos desventajas. Primero, los procesos deben alternarse estrictamente en el uso de su sección crítica; por tanto, el ritmo de ejecución viene dictado por el proceso más lento. Si P0 utiliza su sección crítica sólo una vez por hora, pero P1 desea utilizar su sección crítica a una ratio de 1000 veces por hora, P1 está obligado a seguir el ritmo de P0. Un problema mucho más serio es que si un proceso falla, el otro proceso se encuentra permanentemente bloqueado. Esto se cumple tanto si un proceso falla en su sección crítica como fuera de ella.

La construcción precedente es una **corrutina.** Las corrutinas se diseñan para poder pasar el control de ejecución entre ellas mismas (*véase* el Problema 5.1). Mientras que esto es una técnica útil para un único proceso, es inadecuado para dar soporte a procesamiento concurrente.

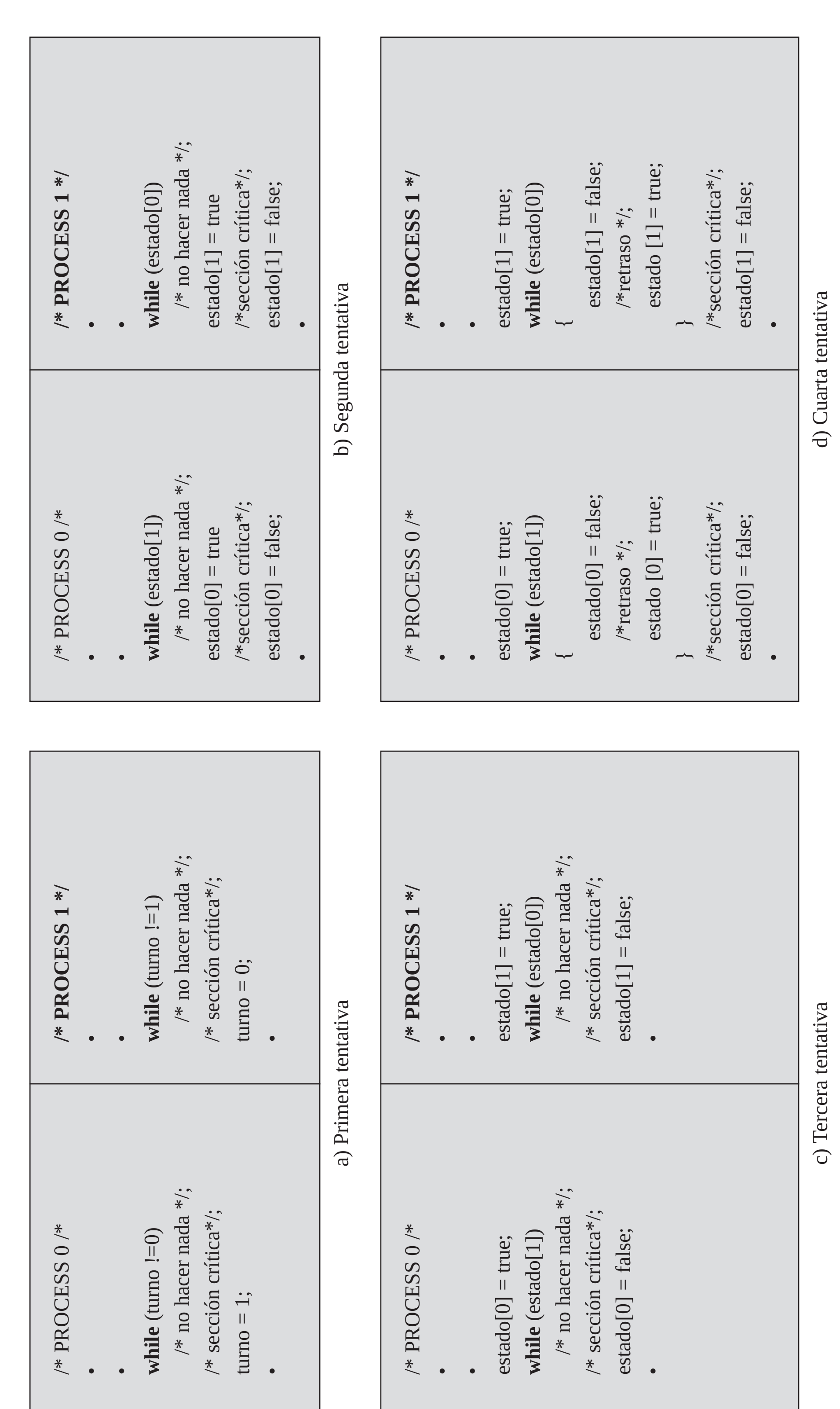


Figura A1. Tentativas de exclusión mutua.

Segunda tentativa

El problema de la primera tentativa es que almacena el nombre del proceso que puede entrar en su sección crítica, cuando de hecho se necesita información de estado sobre ambos procesos. En efecto, cada proceso debería tener su propia llave para entrar en la sección crítica de forma que si uno falla, el otro pueda continuar accediendo a su sección crítica. Para alcanzar este requisito, se define un vector booleano `estado`, con `estado[0]` para P0 y `estado[1]` para P1. Cada proceso puede examinar el estado del otro proceso pero no alterarlo. Cuando un proceso desea entrar en su sección crítica, periódicamente comprueba el estado del otro hasta que tenga el valor `false`, lo que indica que el otro proceso no se encuentra en su sección crítica. El proceso que está realizando la comprobación inmediatamente establece su propio estado a `true` y procede a acceder a su sección crítica. Cuando deja su sección crítica, establece su estado a `false`,

La variable[1] global compartida es ahora:

enum boolean (false=0; true=1);
boolean estado[2]={0,0}

La Figura A.1b muestra el algoritmo. Si un proceso falla fuera de la sección crítica, incluyendo el código de establecimiento de estado, el otro proceso no se queda bloqueado. De hecho, el otro proceso puede entrar en su sección crítica tan frecuentemente como desee, dado que su estado es siempre falso. Sin embargo, si un proceso falla dentro de su sección crítica o después de establecer su estado a verdadero justo antes de entrar en su sección crítica, el otro proceso queda permanentemente bloqueado.

Esta solución es incluso peor que la primera tentativa, ya que no garantiza exclusión mutua en todas las situaciones. Considérese la siguiente secuencia:

P0 ejecuta la sentencia `while` y encuentra `estado[1]` con valor falso.
P1 ejecuta la sentencia `while` y encuentra `estado[0]` con valor falso.
P0 establece `estado[0]` a verdadero y entra en su sección crítica.
P1 establece `estado[1]` a verdadero y entra en su sección crítica.

Debido a que ambos procesos se encuentran ahora en sus secciones críticas, el programa es incorrecto. El problema es que la solución propuesta no es independiente de las velocidades relativas de ejecución de los procesos.

Tercera tentativa

La segunda tentativa falla debido a que un proceso puede cambiar su estado después de que otro proceso lo haya cambiado pero antes de que otro proceso pueda entrar en su sección crítica. Tal vez se pueda arreglar este problema con un simple intercambio de dos sentencias, tal como se muestra en la Figura A.1c.

Como en el caso anterior, si un proceso falla dentro de su sección crítica, incluyendo el código de establecimiento de estado que controla la sección crítica, el otro proceso se bloquea, y si un proceso falla fuera de su sección crítica, el otro proceso no se bloquea.

---

[1] La declaración **enum** se utiliza aquí para declarar un tipo de datos (**boolean**) y asignar sus valores.

A continuación, se va a comprobar que se garantiza la exclusión mutua, desde el punto de vista del proceso P0. Una vez que P0 ha establecido `estado[0]` a verdadero, P1 no puede entrar en su sección crítica hasta que P0 haya entrado y abandonado su sección crítica. Podría ocurrir que P1 ya esté en su sección crítica cuando P0 establece su estado. En dicho caso, P0 quedará bloqueado por la sentencia `while` hasta que P1 haya dejado su sección crítica. El mismo razonamiento se aplica desde el punto de vista de P1.

Esto garantiza la exclusión mutua pero crea otro nuevo problema. Si ambos procesos establecen su estado a verdadero antes de que se haya ejecutado la sentencia `while,` cada uno de los procesos pensará que ha entrado en su sección crítica, causando un interbloqueo.

Cuarta tentativa

En la tercera tentativa, un proceso establece su estado sin conocer el estado del otro proceso. El interbloqueo existe porque cada proceso puede insistir en su derecho a entrar en su sección crítica; no hay oportunidad de retroceder en esta posición. Se puede intentar arreglar este problema de una forma que hace a cada proceso más respetuoso: cada proceso establece su estado para indicar su deseo de entrar en la sección crítica pero está preparado para cambiar su estado si otro desea entrar, tal como se muestra en la Figura A.1d.

Esto está cercano a una solución correcta, pero todavía falla. La exclusión mutua está garantizada, siguiendo un razonamiento similar al usado en la discusión de la tercera tentativa. Sin embargo, considérese la siguiente secuencia de eventos:

P0 establece `estado[0]` a verdadero.
P1 establece `estado[1]` a verdadero.
P0 comprueba `estado[1]`.
P1 comprueba `estado[0]`.
P0 establece `estado[0]` a falso.
P1 establece `estado[1]` a falso.
P0 establece `estado[0]` a verdadero.
P1 establece `estado[1]` a verdadero.

Esta secuencia se podría extender de forma indefinida, y ningún proceso podría entrar en su sección crítica. Estrictamente hablando, esto no es interbloqueo, porque cualquier alteración en la velocidad relativa de los dos procesos rompería este ciclo y permitiría a uno de ellos entrar en su sección crítica. Esta condición se conoce como **círculo vicioso**. Recuérdese que el interbloqueo se produce cuando un conjunto de procesos desea entrar en sus secciones críticas pero ningún proceso puede lograrlo. Cuando se da un círculo vicioso, hay posibles secuencias de ejecución que podrían permitir que se avanzara, pero también es posible describir una o más secuencias de ejecución en las cuales ningún proceso entrara en su sección crítica.

Aunque el escenario descrito es improbable que se mantenga durante mucho tiempo, es no obstante un escenario posible. Por tanto, se debe rechazar la cuarta alternativa.

Una solución válida

Es necesario observar el estado de ambos procesos, lo que se consigue mediante la variable `estado.` Pero, como la cuarta alternativa muestra, esto no es suficiente. Se debe imponer un orden en las acti-

vidades de los dos procesos para evitar el problema de «cortesía mutua» que se ha descrito anteriormente. Se puede utilizar la variable `turno` de la primera alternativa para este propósito; en este caso, la variable indica qué proceso tiene el derecho a insistir en entrar en su región crítica.

Esta solución, conocida como Algoritmo de Dekker, se describe a continuación. Cuando P0 quiere entrar en su sección crítica, establece su estado a verdadero. Entonces comprueba el estado de P1. Si dicho estado es falso, P0 inmediatamente entra en su sección crítica. En otro caso, P0 consulta `turno`. Si encuentra que `turno = 0`, entonces sabe que es su turno para insistir y periódicamente comprueba el estado de P1. P1 en algún punto advertirá que es su turno para permitir al otro proceso entrar y pondrá su estado a falso, provocando que P0 pueda continuar. Después de que P0 ha utilizado su sección crítica, establece su estado a falso para liberar la sección crítica y pone el turno a 1 para transferir el derecho de insistir a P1.

La Figura A.2 proporciona una especificación del Algoritmo de Dekker. La construcción **paralelos**(P1, P2, …, Pn) significa lo siguiente: suspender la ejecución del programa principal; iniciar la ejecución concurrente de los procedimientos P1, P2, … , Pn; cuando todos los procedimientos P1, P2, …, Pn hayan terminado, continuar el programa principal. Se deja una verificación del Algoritmo de Dekker como ejercicio (*véase* Problema A.1).

ALGORITMO DE PETERSON

El Algoritmo de Dekker resuelve el problema de exclusión mutua pero con un programa bastante complejo, que es difícil de seguir y cuya corrección es difícil de probar. Peterson [PETE81] ha proporcionado una solución simple y elegante. Como en el caso anterior, la variable global `estado` indica la posición de cada proceso con respecto a la exclusión mutua y la variable global `turno` resuelve conflictos simultáneos. El algoritmo se presenta en la Figura A.3.

El hecho de que la exclusión mutua se preserva es fácilmente demostrable. Considérese el proceso P0. Una vez que pone su `estado[0]` a verdadero, P1 no puede entrar en su sección crítica. Por tanto, `estado[1] = true` y P0 se bloquea sin poder entrar en su sección crítica. Por otro lado, se evita el bloqueo mútuo. Supóngase que P0 se bloquea en su bucle `while`. Esto significa que `estado[1]` es verdadero y `turno = 1`. P0 puede entrar en su sección crítica cuando `estado[1]` se vuelva falso o el turno se vuelva 0. Ahora considérese los tres casos exhaustivos:

1. P1 no tiene interés en su sección crítica. Este caso es imposible, porque implica que `estado[1] = false`.
2. P1 está esperando por su sección crítica. Este caso es imposible, porque `turn = 1`, P1 es capaz de entrar en su sección crítica.
3. P1 está utilizando su sección crítica repetidamente y por tanto, monopolizando su acceso. Esto no puede suceder, porque P1 está obligado a dar una oportunidad a P0 estableciendo el turno a 0 antes de cada intento por entrar a su sección crítica.

Por tanto, se trata de una solución sencilla al problema de exclusión mutua para dos procesos. Más aún, el Algoritmo de Peterson se puede generalizar fácilmente al caso de *n* procesos [HOFR90].

A2. CONDICIONES DE CARRERA Y SEMÁFOROS

Aunque la definición de una condición de carrera, dada en la Sección 5.1, parece sencilla, la experiencia ha mostrado que los estudiantes normalmente tienen dificultades para señalar las condiciones

```
boolean estado[2];
int turno;
void P0( )
{
      while (true)
      {
            estado[0] = true;
            while (estado [1]
                  if (turno == 1)
            {
                  estado[0] = false;
                  while (turno == 1)
                        /* no hacer nada */;
                  estado[0] = true;
            }
            /* sección crítica */;
            turno = 1;
            estado[0] = false;
            /* resto */;
      }
}
void P1( )
{
      while (true)
      {
            estado[1] = true;
            while (estado[0])
            if (turno == 0)
            {
                  estado[1] = false;
                  while (turno == 0)
                        /* no hacer nada */;
                  estado[1] = true;
            }
            /* sección crítica */;
            turno = 0;
            estado[1] = false;
            /* remainder */;
      }
}
void main ( )
{
      estado[0] = false;
      estado[1] = false;
      turno = 1;
      paralelos (P0, P1);
}
```

Figura A2. Algoritmo de Dekker.

```
boolean estado[2];
int turno;
void P0( )
{
        while (true)
        {
                estado[0] = true;
                turno = 1;
                while (estado [1] && turno == 1)
                        /* no hacer nada */;
                /* sección crítica */;
                estado [0] = false;
                /* resto */;
        }
}
void P1( )
{
        while (true)
        {
                estado [1] = true;
                turno = 0;
                while (estado [0] && turno == 0)
                        /* no hacer nada */;
                /* sección crítica */;
                estado [1] = false;
                /* resto */
        }
}
void main( )
{
        estado [0] = false;
        estado [1] = false;
        paralelos (P0, P1);
}
```

Figura A3. Algoritmo de Peterson para dos procesos.

de carrera de sus programas. El propósito de esta sección, que se basa en [CARR01][2], consiste en dar una serie de pasos a través de varios ejemplos que utilizan semáforos para clarificar el tema de las condiciones de carrera.

[2] Quisiera dar las gracias al profesor Ching-Kuang Shene de la Universidad Tecnológica de Michigan, por permitir utilizar este ejemplo en el libro.

DEFINICIÓN DEL PROBLEMA

Sean dos procesos, **A** y **B**, cada uno de los cuales está compuesto por varios hilos concurrentes. Cada hilo incluye un bucle infinito en el que se intercambia un mensaje con un hilo de otro proceso. Cada mensaje está formado por un entero colocado en un *buffer* global compartido. Hay dos requisitos:

1. Después de que un hilo A1 de un proceso **A** pone disponible un mensaje a algún hilo B1 del proceso **B**, A1 sólo puede continuar si recibe un mensaje de B1. Análogamente, después de que B1 pone un mensaje disponible para A1, sólo puede continuar después de recibir un mensaje de A1.
2. Una vez que un hilo A1 pone un mensaje a disposición, debe asegurarse de que ningún otro hilo de **A** sobrescribe el *buffer* global antes de que un hilo de **B** tome el mensaje.

En el resto de esta sección, se mostrarán cuatro tentativas a la implementación de este esquema utilizando semáforos, cada una de las cuales provoca una condición de carrera. Finalmente, se mostrará una solución correcta.

PRIMERA ALTERNATIVA

Considérese esta opción:

```
semáforo a = 0, b = 0
int buf_a, buf_b;
```

```
hilo_A(…)
{
    int var_a;
    …
    while (true)    {
        …
        var_a = …;
        semSignal(b);
        semWait(a);
        buf_a = var_a;
        var_a = buf_b;
    }   …;
}
```

```
hilo_B(…)
{
    int var_b;
    …
    while (true)    {
        …
        var_b = …;
        semSignal(a);
        semWait(b);
        buf_b = var_b;
        var_b = buf_a;
    }   …;
}
```

Este es un protocolo de *handshaking* (apretón de manos). Cuando un hilo A1 de **A** está listo para intercambiar mensajes, envía una señal a un hilo de **B** y entonces espera a que un hilo B1 de **B** esté listo. Una vez que vuelve una señal desde B1, que A percibe a través de la función ***semWait(a)***, A1 asume que B1 está listo y lleva a cabo el intercambio. B1 se comporta de forma similar, y el intercambio sucede sin importar qué hilo está listo primero.

Esta alternativa puede llevar a condiciones de carrera. Por ejemplo, considérese la siguiente secuencia, sucediéndose en el tiempo de forma vertical:

Hilo A1	Hilo B1
semSignal(b)	
semWait(a)	
	semSignal(a)
	semWait(b)
buf_a = var_a	
var_a = buf_b	
	buf_b = var_b

En la secuencia anterior, A1 alcanza `semWait(a)` y se bloquea. B1 alcanza `semWait(b)` y no se bloquea, pero es expulsado antes de que pueda actualizar su `buf_b`. Mientras tanto, A1 ejecuta y lee de `buf_b` antes de que tenga el valor pretendido. En este punto, `buf_b` tiene un valor proporcionado por otro hilo o proporcionado por B1 en un intercambio anterior. Esto es una condición de carrera.

Una condición de carrera más sutil se puede dar si dos hilos de **A** y **B** están activos. Considérese la siguiente secuencia:

Hilo A1	Hilo A2	Hilo B1	Hilo B2
semSignal(b)			
semWait(a)			
		semSignal(a)	
		semWait(b)	
	semSignal(b)		
	semWait(a)		
		buf_b = var_b1	
			semSignal(a)
buf_a = var_a1			
	buf_a = var_a2		

En esta secuencia, los hilos A1 y B1 intentan intercambiar mensajes y lo hacen a través de las apropiadas instrucciones de señalización de semáforos. Sin embargo, inmediatamente después de que se ejecuten las dos señales `semWait` (en los hilos A1 y B1), el hilo A2 ejecuta `semSignal(b)` y `semWait(a)`, lo que provoca que el hilo B2 ejecute `semSignal(a)` para liberar A2 de `semWait(a)`. En este punto, tanto A1 como A2 podría actualizar `buf_a` a continuación y se daría una condición de carrera. Cambiando la secuencia de ejecución entre los hilos, fácilmente se pueden encontrar otras condiciones de carrera.

Lección aprendida. Cuando se comparta una variable entre múltiples hilos, es probable que se den condiciones de carrera a menos que se utilice protección de exclusión mutua apropiada.

SEGUNDA ALTERNATIVA

En este caso se va a utilizar un semáforo para proteger la variable compartida. El propósito es asegurar que el acceso a `buf_a` y a `buf_b` es mutuamente exclusivo. El programa es el siguiente:

```
semáforo a = 0, b = 0; mutex = 1;
int buf_a, buf_b;
```

```
hilo_A(…)
{
    int var_a;
    …
    while (true)   {
        …
        var_a = …;
        semSignal(b);
        semWait(a);
            semWait(mutex);
                buf_a = var_a;
            semSignal(mutex);
        semSignal(b);
        semWait(a);
            semWait(mutex);
                var_a = buf_b;
            semSignal(mutex);
        …
    }
}
```

```
hilo_B(…)
{
    int var_b;
    …
        while (true)   {
        …
        var_b = …;
        semSignal(a);
        semWait(b);
            semWait(mutex);
                buf_b = var_b;
            semSignal(mutex);
        semSignal(a);
        semWait(b);
            semWait(mutex);
                var_b = buf_a;
            semSignal(mutex);
        …
    }
}
```

Antes de que un hilo pueda intercambiar un mensaje, se lleva a cabo el mismo protocolo de *handshaking* de la primera alternativa. El semáforo `mutex` protege `buf_a` y `buf_b` en un intento de asegurar que la actualización preceda a la lectura. Pero la protección no es adecuada. Una vez que ambos hilos han completado la fase de *handshaking*, los valores de los semáforos `a` y `b` son ambos 1. Se pueden dar entonces tres posibilidades:

1. Dos hilos, digamos A1 y B1, completan el primer *handshaking* y continuan con la segunda fase de intercambio.
2. Otro par de hilos comienzan la primera fase.
3. Un hilo del par actual continuará e intercambiará un mensaje con un hilo recién llegado del otro par.

Todas estas posibilidades pueden llevar a condiciones de carrera. Como un ejemplo de una condición de carrera, basada en la tercera posibilidad, se considera la siguiente secuencia:

Hilo A1	**Hilo A2**	**Hilo B1**
semSignal(b)		
semWait(a)		
		semSignal(a)
		semWait(b)
buf_a = var_a1		
		buf_b = var_b1
	semSinal(b)	
	semWait(a)	
		semSignal(a)
		semWait(b)
	buf_a = var_a2	

En este ejemplo, después de que A1 y B1 realizan el primer *handshake*, ambos actualizan los *buffers* globales correspondientes. Entonces A2 inicia la primera fase de *handshaking*. A continuación, B1 inicia la segunda fase de *handshaking*. En este punto A2 actualiza `buf_a` antes de que B1 pueda leer el valor colocado en `buf_a` por A1. Esto es una condición de carrera.

Lección aprendida. Proteger una única variable puede ser insuficiente si el uso de dicha variable es parte de una larga secuencia de ejecución. Se debe proteger la secuencia de ejecución completa.

TERCERA ALTERNATIVA

Para este intento, se desea expandir la sección crítica para incluir el intercambio de mensajes completo (dos hilos que actualizan uno de dos *buffers* y leen del otro *buffer*). Un único semáforo es insuficiente porque esto podría llevar a interbloqueo, con cada parte esperando por la otra. El programa es el siguiente:

```
semáforo alisto = 1, ahecho = 0, blisto = 1 bhecho = 0
int buf_a, buf_b;
```

```
hilo_A(…)
{
    int var_a;
    …
    while (true)    {
        …
        var_a = …;
        semWait(alisto);
            buf_a = var_a;
            semSignal(ahecho);
            semWait(bhecho);
            var_a = buf_b;
        semSignal(alisto);
        …;
    }
}
```

```
hilo_B(…)
{
    int var_b;
    …
    while (true)    {
        …
        var_b = …;
        semWait(blisto);
            buf_b = var_b;
            semSignal(bhecho);
            semWait(ahecho);
            var_b = buf_a;
            semSignal(blisto);
        …;
    }
}
```

El semáforo `alisto` se utiliza para asegurar que ningún otro hilo de A pueda actualizar `buf_a` mientras que A entra en su sección crítica. El semáforo `ahecho` se utiliza para asegurar que ningún otro hilo de B intente leer `buf_a` hasta que `buf_a` no se haya actualizado. La misma consideración se aplica a `blisto` y `bhecho`. Sin embargo, este esquema no previene las condiciones de carrera. Considérese la siguiente secuencia:

Hilo A1	Hilo B1
buf_a = var_a	
semSignal(ahecho)	
semWait(bhecho)	
	buf_b = var_b
	semSignal(bdone)
	semWait(adone)
var_a = buf_b;	
semSignal(alisto)	
…loop back…	
semWait(alisto)	
buf_a = var_a	
	var_b = buf_a

En esta secuencia, tanto A1 como B1 entran en sus secciones críticas, depositan sus mensajes y alcanzan la segunda espera. Entonces A1 copia el mensaje de B1 y abandona su sección crítica. En este punto, A1 podría volver a su programa, generar un nuevo mensaje y depositarlo en `buf_a`, como se muestra en la secuencia de ejecución anterior. Otra posibilidad es que en ese mismo punto otro hilo de **A** generara un mensaje y lo pusiera en `buf_a`. En este caso, se pierde un mensaje y se produce una condición de carrera.

Lección aprendida. Si tenemos varios grupos de hilos cooperando, la exclusión mutua garantizada para un grupo no puede prevenir de la interferencia de los hilos de otros grupos. Más aún, si un hilo entra repetidamente en una sección crítica, se debe gestionar apropiadamente la temporización de la cooperación entre los hilos.

CUARTA ALTERNATIVA

La tercera alternativa falla al forzar a un hilo a permanecer en su sección crítica hasta que otro hilo recibe el mensaje. Aquí se presenta un intento para lograr este objetivo:

```
semáforo alisto = 1, ahecho = 0, blisto = 1 bhecho = 0
int buf_a, buf_b;
```

```
hilo_A(…)
{
    int var_a;
    …
    while (true)   {
        …
        var_a = …;
        semWait(blisto);
            buf_a = var_a;
            semSignal(ahecho);
            semWait(bhecho);
            var_a = buf_b;
        semSignal(alisto);
        …;
    }
}
```

```
hilo_B(…)
{
    int var_b;
    …
    while (true)   {
        …
        var_b = …;
    semWait(alisto);
        buf_b = var_b;
        semSignal(bhecho);
        semWait(ahecho);
        var_b = buf_a;
    semSignal(blisto);
    …;
    }
}
```

En este caso, el primer hilo de **A** que entra en su sección crítica decrementa `blisto` a 0. Ningún hilo de **A** intentará un intercambio de mensajes hasta que un hilo de **B** complete el intercambio de mensajes e incremente `bready` a 1. Esta técnica también puede llevar a condiciones de carrera, tales como en la siguiente secuencia:

Hilo A1	**Hilo A2**	**Hilo B1**
semWait(blisto)		
buf_a = var_a1		
semSignal(ahecho)		
		semWait(alisto)
		buf_b = var_b1
		semSignal(bhecho)
		semWait(ahecho)
		var_b = buf_a
		semSignal(blisto)
	semWait(blisto)	
	• • •	
	semWait(bhecho)	
	var_a2 = buf_b	

En esta secuencia, los hilos A1 y B1 entran en las correspondientes secciones críticas a fin de intercambiar mensajes. El hilo B1 recupera su mensaje y señaliza `blisto.` Esto permite a otro hilo de **A**, A2, entrar en su sección crítica. Si A2 es más rápido que A1, entonces A2 puede recuperar el mensaje enviado para A1.

Lección aprendida. Se pueden dar condiciones de carrera si el semáforo de exclusión mutua no es liberado por su propietario. En esta cuarta alternativa, un hilo de **A** bloquea un semáforo y un hilo de **B** lo desbloquea. Ésta es una práctica de programación arriesgada.

UNA ALTERNATIVA CORRECTA

El lector puede notar que el problema de esta sección es una variación del problema del *buffer* acotado y por tanto, se puede resolver de una forma similar a la discusión de la Sección 5.4. La técnica más directa es utilizar dos *buffers*, uno para los mensajes de B hacia A y otro para los mensajes de A hacia B. El tamaño de cada *buffer* necesita ser igual a uno. Para entender las razones de esto, considérese que no hay ninguna suposición de orden para la liberación de los hilos desde una primitiva de sincronización. Si un *buffer* tiene más de una entrada, entonces no se puede garantizar que los mensajes los reciba el destinatario apropiado. Por ejemplo, B1 podría recibir un mensaje de A1 y entonces enviar un mensaje a A1. Pero si el *buffer* tiene múltiples entradas, otro hilo de **A** podría adquirir el mensaje de la entrada que corresponde a A1.

Utilizando el mismo enfoque básico que se utiliza en la Sección 5.4, podemos desarrollar el siguiente programa:

```
semáforo notFull_A = 1, notFull_B = 1
semáforo notEmpty_A = 0, notEmpty_B = 0;
int buf_a, buf_b;
```

```
hilo_A(…)
{
    int var_a;
    …
    while (true)   {
        …
        var_a = …;
        semWait(notFull_A);
            buf_a = var_a;
            semSignal(notEmpty_A);
        semWait(notEmpty_B);
            var_a = buf_b;
            semSignal(notFull_B);
        …;
    }
}
```

```
hilo_B(…)
{
    int var_b;
    …
    while (true)   {
        …
        var_b = …;
        semWait(notFull_B);
            buf_b = var_b;
            semSignal(notEmpty_B);
        semWait(notEmtpy_A);
            var_b = buf_a;
            semSignal(notFull_A);
        …;
    }
}
```

Para verificar que esta solución funciona, se necesita tratar tres aspectos:

1. La sección de intercambio de mensajes es mutuamente exclusiva dentro del grupo de hilos. Debido a que el valor inicial de `nolleno_A` es 1, sólo un hilo de **A** puede pasar a través de ***semWait(noLleno_A)*** hasta que el intercambio se haya completado y lo haya señalizado algún hilo de **B** que ejecute ***semSignal(noLleno_A)***. Un razonamiento similar se aplica a los hilos en **B**. Por tanto, se cumple esta condición.

2. Una vez que dos hilos entran en sus secciones críticas, intercambian mensajes sin interferirse entre sí. Ningún otro hilo de **A** puede entrar a su sección crítica hasta que el hilo de **B** haya finalizado completamente el intercambio, y ningún otro hilo de **B** puede entrar en su sección crítica hasta que el hilo de **A** haya finalizado completamente el intercambio. Por tanto, se cumple esta condición.
3. Después de que un hilo abandona su sección critica, ningún otro hilo del mismo grupo puede darse prisa y estropear el mensaje existente. Esta condición se satisface por el hecho de utilizar un *buffer* de una única entrada para cada dirección. Una vez que un hilo de **A** ha ejecutado ***semWait(noLleno_A)*** y ha entrado en su sección crítica, ningún otro hilo de **A** puede actualizar `buf_a` hasta que el correspondiente hilo de **B** haya recuperado el valor de `buf_a` y realizado un ***semSignal(noLleno_A)***.

Lección aprendida: Es bueno revisar las soluciones a problemas conocidos, porque una solución correcta a un problema dado puede ser una variación de una solución a un problema conocido.

A3. EL PROBLEMA DE LA BARBERÍA

Como otro ejemplo de uso de semáforos para implementar la concurrencia, considérese el simple problema[3] de la barbería. Este ejemplo es instructivo, porque los problemas encontrados cuando se intenta proporcionar un acceso adaptado a los recursos de la barbería, es similar a aquéllos encontrados en un sistema operativo real.

Nuestra barbería tiene tres sillas, tres barberos, un área de espera que puede acomodar a cuatro clientes en un sofá y una habitación donde clientes adicionales pueden permanecer de pie (Figura A.4). Las normas de incendios limitan el número total de clientes en la tienda a 20. En este ejemplo, se va a asumir que la barbería procesa finalmente 50 clientes.

Un cliente no entrará en la tienda si está llena. Una vez dentro, el cliente toma asiento en el sofá o permanece de pie si el sofá está lleno. Cuando un barbero está libre, sirve al cliente que ha permanecido en el sofá durante un tiempo mayor. Si hay algún cliente de pie, el que lleve más tiempo en la tienda toma asiento en el sofá. Cuando un cliente finaliza, cualquier barbero acepta el pago, pero debido a que sólo hay una máquina registradora, se acepta el pago de un único cliente a la vez. Los barberos dividen su tiempo entre cortar el pelo, aceptar pagos y dormir en su silla cuando esperan por un cliente.

UNA BARBERÍA INJUSTA

La Figura A.5 muestra una implementación que utiliza semáforos; los tres procedimientos se listan en columnas para ahorrar espacio. Se asume que todas las colas de semáforos se gestionan con una política FIFO.

El cuerpo del programa principal activa 50 clientes, 3 barberos y el proceso cajero. Ahora se considerará el propósito y la colocación de los diferentes operadores de sincronización:

- **Capacidad de la tienda y del sofá.** La capacidad de la tienda y la capacidad del sofá se gobiernan a través de los semáforos `max_capacidad` y `sofa`, respectivamente. Cada vez que

[3] Estoy en deuda con el profesor Ralph Hilzer de la Universidad del Estado de California en Chico, por proporcionarme este tratamiento del problema.

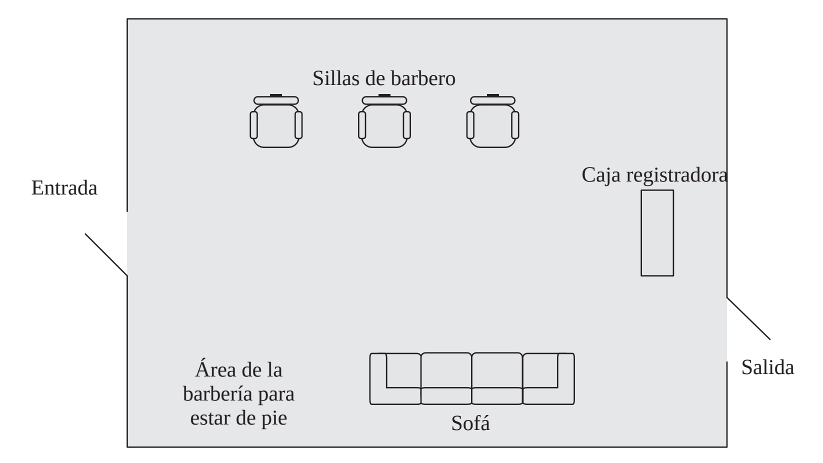


Figura A4. La barbería.

un cliente intenta entrar en la tienda, el semáforo `max_capacidad` se decrementa en 1; cada vez que un cliente abandona la barbería, el semáforo se incrementa. Si un cliente encuentra la tienda llena, el proceso de dicho cliente se bloquea en `max_capacidad` mediante la función ***semWait***. Análogamente, las operaciones ***semWait*** y ***semSignal*** están alrededor de las acciones de sentarse y levantarse del sofá.

- **Capacidad de la silla del barbero.** Hay tres sillas de barbero y hay que tener cuidado con utilizarlas apropiadamente. El semáforo `silla_barbero` asegura que no hay más de tres clientes intentando obtener servicio a la vez, intentando evitar la indigna situación de tener un cliente sentado en el regazo de otro. Un cliente no se levantará del sofá, a menos que una silla esté libre [***semWait(silla_barbero)***] y cada barbero señaliza la acción de que un cliente ha dejado su silla [***semSignal(silla_barbero)***]. Se asegura el acceso justo a las sillas de los barberos debido a la organización de cola del semáforo: el primer cliente que se bloquea será el primero que ocupe una silla disponible. Obsérvese que, en el procedimiento del cliente, si ***semWait(silla_barbero)*** ocurriese después de ***semSignal(sofa)***, cada cliente brevemente se sentaría en el sofá y permanecería de pie en la línea de las sillas de los barberos, creando congestión y dejando a los barberos poco espacio de maniobra.
- **Asegurar que los clientes están en la silla del barbero.** El semáforo `cliente_listo` envía una señal que despierta a un barbero que se encuentre durmiendo, indicándole que un cliente acaba de ocupar una silla. Sin este semáforo, un barbero nunca dormiría sino que empezaría a cortar el pelo tan pronto como un cliente dejara la silla; si no hubiese ningún cliente nuevo sentado, el barbero estaría cortando el aire.
- **Mantenimiento de los clientes en una silla de barbero.** Una vez sentado, un cliente permanece en la silla hasta que el barbero da la señal de que el corte ha finalizado, utilizando el semáforo `terminado`.
- **Limitar un cliente por silla de barbero.** El semáforo `silla_barbero` se utiliza para limitar a tres el número de clientes que hay en las sillas de los barberos. Sin embargo, por sí mismo, el semáforo `silla_barbero` no realiza adecuadamente su misión. Un cliente que no obtiene el procesador inmediatamente después de que su barbero ejecuta ***semSignal(terminado)*** (es decir, que se cae o se para a hablar con un vecino) podría estar todavía en la silla cuando al próximo cliente se le permita sentarse. El semáforo `dejar_silla_b` se utiliza para corregir este problema, evitando que el barbero invite a un nuevo cliente a la silla hasta que el cliente

```
/* programa barberia1 */
semaforo max_capacidad = 20;
semaforo sofa = 4;
semaforo silla_barbero = 3;
semaforo coord = 3;
semaforo cliente_listo = 0, terminado = 0, dejar_silla_b = 0; pago = 0; recibo = 0;
void cliente ()
{
    wait(max_capacidad);
    entrar_tienda();
    wait(sofa);
    sentarse_en_sofa();
    wait(silla_barbero);
    levantarse_del_sofa();
    signal(sofa);
    sentarse_en_silla_de_barbero
    signal(cliente_listo);
    wait(terminado);
    dejar_silla_barbero();
    signal(dejar_silla_b);
    pagar();
    signal(pago);
    wait(recibo);
    salir_tienda();
    signal(max_capacidad)
}

void barbero ()
{
    while (true)
    {
        wait(cliente_listo);
        wait(coord);
        cortar_pelo();
        signal(coord);
        signal(terminado);
        wait(dejar_silla_b)
        signal(silla_barbero);
    }
}

void cajero ()
{
    while (true)
    {   wait(pago);
        wait(coord);
        aceptar_pago();
        signal(coord);
        signal(recibo);
    }
}

void main()
{
parbegin (cliente,… 50 veces,… cliente, barbero, barbero, barbero, cajero);
}
```

Figura A5. Una barbería injusta.

no se haya ido efectivamente. En los problemas del final de este capítulo se analizará que incluso esta precaución no permite evitar que un cliente vigoroso se siente en el regazo de un cliente.

- **Pagos y recibos.** Naturalmente, hay que ser cuidadoso cuando se trata de dinero. El cajero debe asegurar que cada cliente paga antes de abandonar la tienda y el cliente quiere la verificación de que se ha recibido el pago (un recibo). Esto se realiza, en efecto, mediante una transferencia monetaria cara a cara. Cada cliente, justo después de levantarse de la silla del barbero, paga, avisando al cajero que se ha pasado el dinero [***semSignal(pago)***] y entonces espera por un recibo [***semWait(recibo)***]. El proceso cajero se dedica de forma repetida a recibir los pagos: espera a que se le señalice un pago, acepta el dinero y entonces señaliza la aceptación del dinero. Aquí se necesitan evitar varios errores de programación. Si ***semSignal(pago)*** ocurriera justo antes de la acción ***pagar***, un cliente podría verse interrumpido después de tal señaliza-

ción; esto dejaría libre al cajero para aceptar pagos incluso de alguien que no se lo hubiera ofrecido. Un error aún más serio sería invertir las posiciones de las sentencias ***semSignal(pago)*** y ***semWait(recibo)***. Esto llevaría a un interbloqueo ya que provocaría que todos los clientes y el cajero se bloquearan en sus respectivos operadores ***semWait***.

- **Coordinación entre las funciones de barbero y de cajero.** Para ahorrar dinero, esta barbería no emplea un cajero independiente. Cada barbero puede llevar a cabo esta tarea si no está cortando el pelo. El semáforo coord asegura que sólo un barbero lleva a cabo esta tarea en un momento determinado.

La Tabla A.1 resume el uso de cada uno de los semáforos del programa.

Tabla A.1. Funcionalidad de los semáforos en la Figura A.5.

Semáforo	Operación *wait*	Operación *signal*
`max_capacidad`	El cliente espera a que haya espacio en la tienda.	Un cliente que abandona la tienda señaliza a un cliente que espera por entrar.
`sofa`	El cliente espera para sentarse en el sofá.	Un cliente que abandona el sofa señaliza a un cliente que espera por sentarse en el sofá.
`silla_barbero`	El cliente espera por una silla de barbero vacía.	El barbero señaliza cuando su silla está vacía.
`cliente_listo`	El barbero espera a que el cliente se siente en la silla.	El cliente señaliza al barbero para indicarle que está en la silla.
`terminado`	El cliente espera hasta que se completa su corte de pelo.	El barbero señaliza cuando ha finalizado de cortar el pelo a su cliente.
`dejar_silla_b`	El barbero espera hasta que el cliente se levanta de la silla.	El cliente señaliza al barbero cuando se ha levantado de la silla.
`pago`	El cajero espera a que el cliente pague.	El cliente señaliza al cajero que ha pagado.
`recibo`	El cliente espera un recibo de su pago.	El cajero señaliza que se ha aceptado el pago.
`coord`	Espera por un barbero que esté libre para llevar a cabo un corte de pelo o tareas de cajero.	Señaliza que un barbero está libre.

El proceso cajero se podría eliminar, mezclando la función de pago en el procedimiento del barbero. Cada barbero secuencialmente cortaría el pelo y a continuación aceptaría el pago. Sin embargo, con una única caja registradora, es necesario limitar el acceso a la función de pago aceptar a un único barbero en un momento determinado. Esto se podría hacer tratando la función como una sección crítica y protegiéndola con un semáforo.

UNA BARBERÍA JUSTA

La Figura A.5 es un buen intento, pero quedan algunos problemas. El resto de esta sección se encarga de resolver uno de estos problemas; otros se dejan como ejercicios para el lector (*véase* Problema A.6).

```
/* programa barberia2 */
semaforo max_capacidad = 20;
semaforo sofa = 4;
semaforo silla_barbero = 3, coord = 3;
semaforo mutex1 = 1, mutex2 = 1;
semaforo cliente_listo = 0, dejar_silla_b = 0, pago = 0; recibo = 0;
semaforo terminado [50] = {0};
int count;

void cliente ()
{
    int numcliente;
    wait(max_capacidad)
    Entrar_tienda();
    wait(mutex1);
    numcliente = cuenta;
    cuenta++;
    signal(mutex1);
    wait(sofa);
    sentarse_en_sofa();
    wait(silla_barbero);
    levantarse_del_sofa();
    signal(sofa);
    sentarse_en_silla_de_barbero();
    wait(mutex2);
    encola1(numcliente);
    signal(cliente_listo);
    signal(mutex2);
    wait(terminado[numcliente]);
    dejar_silla_barbero();
    signal(dejar_silla_b);
    pagar();
    signal(pago);
    wait(recibo);
    salir_tienda();
    signal(max_capacidad)
}
```

```
void barbero ()
{
    int cliente_b
    while (true)
    {
        wait(cliente_listo);
        wait(mutex2);
        fcola1(cliente_b);
        signal(mutex2);
        wait(coord);
        cortar_pelo();
        signal(coord);
        signal(terminado[cliente_b]);
        wait(dejar_silla_b);
        signal(silla_barbero);
    }
}
```

```
void cajero ()
{
    while (true)
    {
        wait(pago);
        wait(coord);
        aceptar_pago();
        signal(coord);
        signal(recibo);
    }
}
```

```
void main ()
{   count :=0;
    parbegin (cliente,… 50 veces,… cliente, barbero, barbero, barbero, cajero);
}
```

Figura A6. Una barbería justa.

Hay un problema de temporización en la Figura A.5 que podría llevar a un tratamiento injusto por parte de los clientes. Supóngase que hay actualmente tres clientes sentados en las tres sillas de los barberos. En este caso, lo más probable es que los clientes estén bloqueados en *semWait(terminado)*, y debido a la organización de cola, se liberen en el orden en que se sentaron. Sin embargo, ¿qué pasa si uno de los barberos es muy rápido o uno de los clientes es bastante calvo? Liberando al primer cliente que se sentó podría resultar una situación en la que un cliente fuera echado de su silla y forzado a pagar el precio completo por un corte de pelo parcial mientras otros son retenidos en la silla incluso cuando se haya terminado su corte de pelo.

El problema se resuelve con más semáforos, como se muestra en la Figura A.6. Se asigna un único número de cliente a cada cliente; esto es equivalente a que cada cliente coja un número al entrar en la tienda. El semáforo mutex1 protege el acceso a la variable global cuenta de forma que cada cliente recibe un número único. El semáforo terminado se redefine como un vector de 50 semáforos. Una vez que el cliente se ha sentado en una silla de barbero, ejecuta ***semWait(terminado[numCliente])*** para esperar por su propio semáforo; cuando el barbero finaliza con dicho cliente, ejecuta ***semSignal(terminado[numCliente])*** para liberar al cliente apropiado.

Queda por explicar cómo un barbero conoce el número del cliente. Un cliente coloca su número en la cola encola1 justo antes de señalizar al barbero con el semáforo cliente_listo. Cuando un barbero está listo para cortar el pelo, fcola1(cliente_b) borra el número de cliente que se encuentra en la cima de la cola cola1 y lo coloca en la variable local del barbero cliente_b.

A4. PROBLEMAS

A.1. Demuestre la corrección del Algoritmo de Dekker.

a) Muestre que se fuerza la exclusión mutua. *Pista:* muestre que cuando Pi entra en su sección crítica, se cumple la siguiente expresión:

estado[i] **and** (**not** estado[1-i])

b) Muestre que un proceso que requiere acceso a su sección crítica no es retrasado indefinidamente. *Pista:* considere los siguientes casos: (1) un único proceso está intentando entrar a su sección crítica; (2) ambos procesos están intentando entrar en la sección crítica y (2a) *turno = 0* y *estado[0]* = falso, y (2b) *turno = 0* y *estado[0]* = verdadero.

A.2. Considere el Algoritmo de Dekker, escrito para un número arbitrario de procesos, cambiando la sentencia ejecutada cuando se abandona la sección crítica de:

turno = 1-i; /* es decir, P0 establece turno a 1 y P1 establece turno a 0 */

a

turno = (turno+1)%n; /* n = número de procesos*/

Evalúe el algoritmo cuando el número de procesos que ejecutan concurrentemente es mayor de dos.

A.3. Demuestre que las siguientes técnicas de software para exclusión mutua no dependen de la exclusión mutua elemental a nivel de acceso a memoria:

a) el algoritmo de la panadería

b) el Algoritmo de Peterson

A.4. Desarrolle una solución para el problema discutido en la Sección A.2, en el cual se utiliza un único *buffer* de una entrada. En este caso, los dos hilos deben turnarse para intercambiar sus mensajes en lugar de operar en paralelo.

A.5. Conteste a las siguientes cuestiones relacionadas con la barbería justa (Figura A.6):

a) ¿El código requiere que el barbero que termina de hacer un corte de pelo recoja el pago de dicho cliente?

b) ¿Los barberos siempre utilizan la misma silla?

A.6. Se mantienen varios problemas en el caso de la barbería justa de la Figura A.6. Modifique el problema para corregir los siguientes problemas.

a) El cajero podría aceptar el pago de un cliente y liberar a otro si dos o más clientes están esperando por pagar. Afortunadamente, una vez que un cliente presenta un pago, no hay forma para él de retroceder, así que al final, en el caja registradora se guarda la cantidad exacta. No obstante, es deseable liberar al cliente correcto tan pronto como haya realizado su pago.

b) El semáforo dejar_silla_b evita supuestamente que múltiples clientes accedan a una única silla de barbero. Desafortunadamente, este semáforo no realiza su tarea adecuadamente en todos los casos. Por ejemplo, supóngase que los tres barberos han cortado el pelo y se han quedado bloqueados en ***semWait(dejar_silla_b)***. Dos de los clientes se encuentran en un estado interrumpido justo antes de ***dejar la silla del barbero***. El tercer cliente deja su silla y ejecuta ***semSignal(dejar_silla_b)***. ¿Qué barbero se libera? Debido a que la cola ***dejar_silla_b*** es FIFO, se libera el primer barbero que se bloqueó. ¿Es este barbero el que estaba cortando el pelo del cliente que señalizó? Tal vez, pero tal vez no. Si no es así, un nuevo cliente vendrá y se sentará en el regazo de otro cliente que estaba a punto de levantarse.

c) El programa requiere que un cliente se siente primero en el sofá, incluso si la silla del barbero está vacía. Éste es un problema menor y resolver este problema implica hacer un código aún más complejo. No obstante, intente resolver este problema.

APÉNDICE B

Diseño orientado a objetos

◆ ◆

Windows y otros sistemas operativos de hoy día se fundamentan en los principios del diseño orientado a objetos. Este apéndice proporciona una breve visión general de los principales conceptos del diseño orientado a objetos.

◆ ◆

B.1. MOTIVACIÓN

Los conceptos de orientación a objetos se han hecho bastante populares en el área de la programación, con la promesa de construir fragmentos de software intercambiables, reutilizables, fácilmente actualizables y fácilmente interconectables. Más recientemente, los diseñadores de bases de datos han comenzado a apreciar las ventajas de la orientación a objetos, con el resultado de que han comenzado a aparecer sistemas de gestión de bases de datos orientadas a objetos (OODBMS).

La programación orientada a objetos y los sistemas de gestión de bases de datos orientadas a objetos son, de hecho, cosas diferentes, pero comparten un concepto clave: este software o estos datos pueden ser «alojados en contenedores». Cualquier cosa va dentro de una caja, y puede haber cajas dentro de otras cajas. En el programa convencional más simple, un paso del programa equivale a una instrucción; en un lenguaje orientado a objetos, cada paso puede ser una caja completa de instrucciones. Similarmente, en una base de datos orientada a objetos, una variable, en vez de equivaler a un simple elemento de datos, puede equivaler a una caja completa de datos.

La Tabla B.1 introduce algunos de los términos clave utilizados en el diseño orientado objetos.

Tabla B.1. Términos clave de orientación a objetos.

Término	Definición
Atributo	Variables de datos contenidas dentro de un objeto.
Agregación	Relación entre dos instancias de objeto en la cual el objeto contenedor incluye un puntero al objeto contenido.
Encapsulación	El aislamiento de los atributos y servicios de una instancia del objeto respecto al entorno externo. Los servicios sólo pueden invocarse por nombre y los atributos sólo pueden accederse por medio de servicios.
Herencia	Relación entre dos clases de objetos en la cual los atributos y servicios de la clase padre son adquiridos por la clase hija.
Interfaz	Descripción muy relacionada con una clase objeto. Un interfaz contiene definiciones de métodos (sin implementaciones) y valores constantes. Un interfaz no puede instanciarse como un objeto
Mensaje	El medio mediante el cual los objetos interactúan.
Método	Procedimiento que es parte de un objeto y que puede activarse desde fuera del objeto para realizar ciertas funciones.
Objeto	Abstracción de una entidad del mundo real.
Clase de objeto	Nombre de un conjunto de objetos que comparten los mismos nombres, conjunto de atributos y servicios.
Instancia de objeto	Miembro específico de una clase de objetos, con valores asignados a los atributos.
Polimorfismo	Se refiere a la existencia de múltiples objetos que usan los mismos nombres para servicios y presentan el mismo interfaz al mundo exterior pero que representan diferentes tipos de entidades.
Servicio	Función que realiza una operación en un objeto.

B.2. CONCEPTOS DE ORIENTACIÓN A OBJETOS

El concepto central del diseño orientado a objetos es el de objeto. Un objeto es una unidad software única que contiene una colección de variables relacionadas (datos) y de métodos (procedimientos). Generalmente, estas variables y métodos no son directamente visibles desde fuera del objeto. En cambio, existen interfaces bien definidos que permiten que otro software tenga acceso a los datos y los procedimientos.

Un objeto representa alguna cosa, ya sea una entidad física, un concepto, un módulo software o alguna entidad dinámica como una conexión TCP. Los valores de las variables del objeto expresan la información que se conoce sobre la cosa que el objeto representa. Los métodos incluyen procedimientos cuya ejecución afecta a los valores en el objeto y posiblemente también afectan a la cosa que se representa.

Las Figuras B.1 y B.2 ilustran los conceptos clave de orientación a objetos.

ESTRUCTURA DEL OBJETO

Los datos y procedimientos contenidos en un objeto se conocen normalmente como variables y métodos respectivamente. Todo aquello que un objeto «conoce» puede expresarse en sus variables, y todo aquello que puede hacer se expresa en sus métodos.

Las **variables** de un objeto, también llamadas atributos, generalmente son escalares sencillos o tablas. Cada variable tiene un tipo, posiblemente un conjunto de valores permitidos y puede ser constante



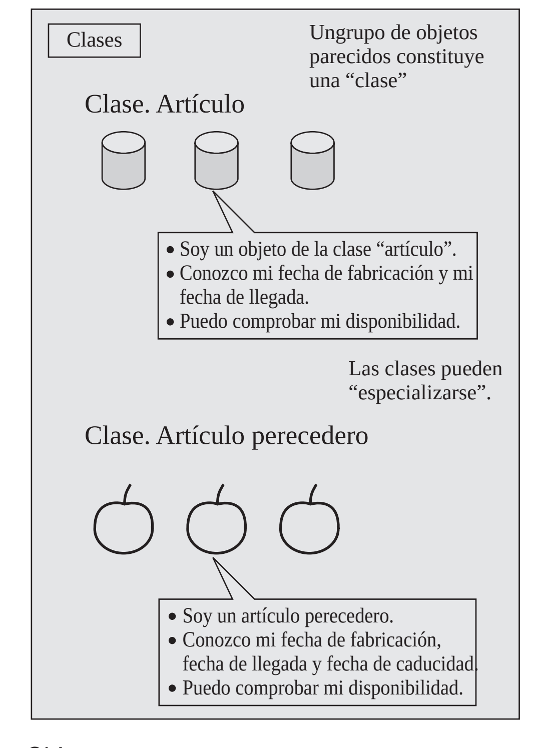
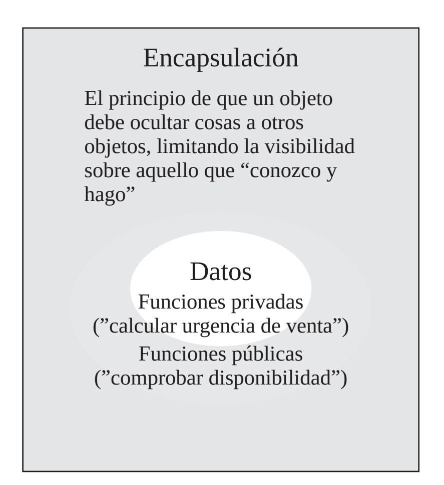
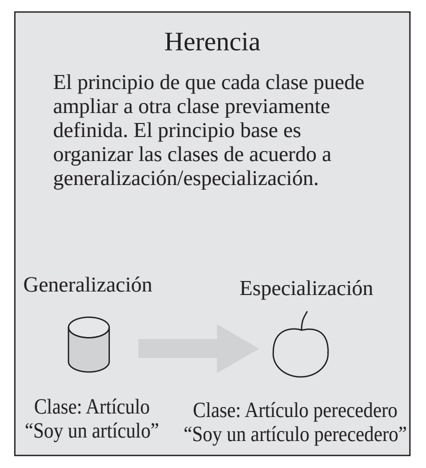


Figura B.1. Objetos.





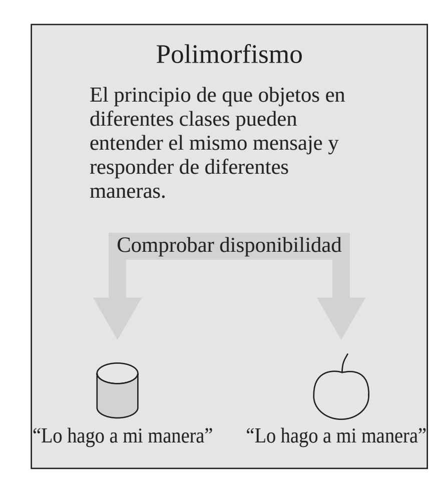


Figura B.2. Conceptos de objetos.

o variable (por convención, el término variable se utiliza también para las constantes). También pueden imponerse restricciones de acceso a las variables para ciertos usuarios, clases de usuarios o situaciones.

Los **métodos** de un objeto son procedimientos que deben dispararse desde fuera para realizar ciertas funciones. El método puede cambiar el estado del objeto, actualizar algunas de sus variables o actuar sobre recursos externos a los cuales el objeto tiene acceso.

Los objetos interactúan por medio de **mensajes**. Un mensaje incluye el nombre del objeto emisor, el nombre del objeto receptor, el nombre del método del objeto receptor y cualesquiera parámetros necesarios para cualificar la ejecución del método. Un mensaje sólo puede usarse para invocar un método de un objeto. La única manera de acceder a los datos internos de un objeto es mediante los métodos del objeto. Así, un método puede hacer que se tome cierta acción o que se accedan las variables del objeto o ambas cosas. Para los objetos locales, pasarle un mensaje a un objeto es lo mismo que llamar a un método del objeto. Cuando los objetos están distribuidos, pasar un mensaje es exactamente lo que la expresión indica.

El interfaz de un objeto es un conjunto de métodos públicos que el objeto proporciona. Un interfaz no dice nada en relación a su implementación; objetos de diferentes clases pueden tener diferentes implementaciones de los mismos interfaces.

La propiedad de que para un objeto, su único interfaz con el mundo exterior es por medio de mensajes se conoce como **encapsulación**. Los métodos y variables de un objeto están encapsulados y disponibles sólo vía comunicación basada en mensajes. La encapsulación ofrece dos ventajas:

1. Protege a las variables de un objeto de corrupción por otros objetos. Esta protección puede incluir protección frente a accesos no autorizados y protección frente a ciertos tipos de problemas que surgen con los accesos concurrentes, como los interbloqueos y valores inconsistentes.
2. Oculta la estructura interna del objeto y así la interacción con el objeto es relativamente simple y estandarizada. Es más, si la estructura interna o los procedimientos de un objeto se modifican sin alterar su funcionalidad externa, otros objetos no se verán afectados.

CLASES DE OBJETOS

En la práctica, normalmente habrá varios objetos representando los mismos tipos de cosas. Por ejemplo, si un proceso se representa con un objeto, entonces habrá un objeto por cada proceso presente en un sistema. Claramente, cada uno de tales objetos necesita su propio conjunto de variables. Sin embargo, si los métodos del objeto son procedimientos reentrantes, los objetos similares podrían compartir los mismos métodos. Es más, sería ineficiente redefinir tanto los métodos como las variables para cada nuevo, pero similar, objeto.

La solución a estas dificultades es distinguir entre una clase de objetos y una instancia de objeto. Una **clase de objetos** es una plantilla que define los métodos y variables que deben incluirse en un tipo de objeto particular. Una **instancia de objeto** es un objeto real que incluye las características de la clase que lo define. La **instanciación** es el proceso de creación de una nueva instancia de objeto de una clase de objetos.

Herencia. El concepto de clase de objetos es potente porque permite la creación de muchas instancias de objeto con un mínimo esfuerzo. Este concepto se hace todavía más potente mediante el uso del mecanismo de herencia [TAIV96].

La herencia permite definir una nueva clase de objetos en términos de una clase existente. La nueva clase (nivel inferior), llamada **subclase** o **clase hija**, automáticamente incluye los métodos y las definiciones de variable de la clase original (nivel superior), llamada **superclase** o **clase padre**. Una subclase puede diferir de su superclase de varias maneras:

1. La subclase puede incluir métodos y variables adicionales que no se encuentran en su superclase.
2. La subclase puede anular la definición de cualquier método o variable de su superclase usando el mismo nombre con una nueva definición. Esto, ofrece una manera simple y eficiente de manipular casos especiales.
3. La subclase puede restringir de alguna manera un método o variable heredados de su superclase.

   La Figura B.3, basada en una de [KORS90], ilustra el concepto.

El mecanismo de herencia es recursivo, permitiendo a una subclase ser superclase de sus propias subclases. De este modo, puede construirse una **jerarquía de herencia**. Conceptualmente, se puede entender que la jerarquía de herencia define una técnica de búsqueda de métodos y variables. Cuando un objeto recibe un mensaje para llevar acabo un método que no está definido en su clase, automáticamente se explora hacia arriba en la jerarquía hasta que se encuentra el método. De igual modo, si la

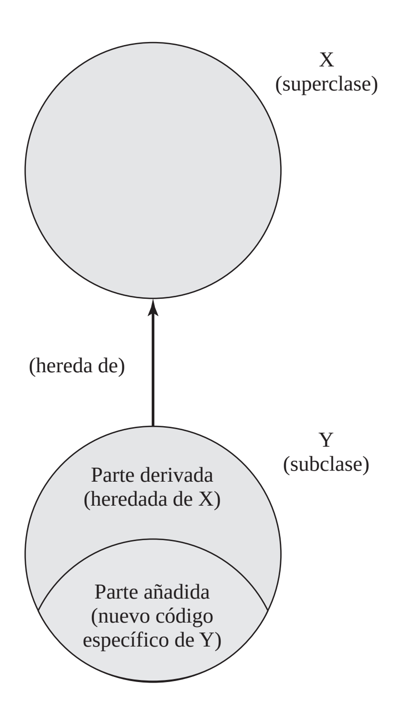


Figura B.3. Herencia.

ejecución de un método referencia una variable que no está definida en esa clase, el objeto se busca en la jerarquía hasta encontrar la variable con dicho nombre.

Polimorfismo. El polimorfismo es una fascinante y potente característica que hace posible ocultar diferentes implementaciones bajo un interfaz común. Dos objetos que son polimórficos entre sí utilizan los mismos nombres para los métodos y presentan el mismo interfaz a otros objetos. Por ejemplo, puede haber varios objetos de impresión, para diferentes dispositivos de salida, tales como imprimirMatricial, imprimirLaser, imprimirPantalla y otros, o para diferentes tipos de documentos, tales como imprimirTexto, imprimirDibujo o imprimirCompuesto. Si cada objeto incluye un método llamado imprimir, entonces podrá imprimirse cualquier documento enviando un mensaje imprimir al objeto apropiado, sin preocuparnos por cómo se lleva a cabo realmente dicho método. Normalmente, el polimorfismo se utiliza para permitir tener el mismo método en múltiples subclases de una misma superclase, cada uno con detalles de implementación distintos.

Es instructivo comparar el polimorfismo con las técnicas usuales de programación modular. Uno de los objetivos del diseño modular de arriba a abajo, es diseñar módulos de nivel inferior de utilidad general con un interfaz fijo frente los módulos de nivel superior. Esto permite que un módulo de nivel inferior se invoque por diferentes módulos de nivel superior. Si se cambian los entresijos de un módulo de nivel inferior sin cambiar su interfaz, ninguno de los módulos de nivel superior que lo utilizan se verán afectados. En cambio, con el polimorfismo, lo que preocupa es la capacidad del objeto de nivel superior de invocar diferentes objetos de nivel inferior utilizando el mismo formato de mensaje para llevar acabo funciones similares. Con el polimorfismo, pueden añadirse nuevos objetos de nivel inferior con cambios mínimos sobre los objetos existentes.

Interfaces. La herencia permite que un objeto de una subclase utilice la funcionalidad de su superclase. Puede haber casos en los que se desee definir una subclase que tenga las funcionalidades de más de una superclase. Esto se puede conseguir permitiendo que una subclase herede de más de una

superclase. C++ es uno de los lenguajes que permite tal herencia múltiple. Sin embargo, por simplicidad, la mayoría de los lenguajes orientados a objetos modernos, incluyendo Java, C# y Visual Basic .NET, limitan la herencia a una única superclase. En cambio, una característica conocida como *interfaces* se utiliza para permitir que una clase tome prestada cierta funcionalidad de una clase y otra funcionalidad de una clase completamente diferente.

Desafortunadamente, el término *interfaz* se utiliza mucho en la literatura de objetos con dos significados uno de propósito general y otro específicamente funcional. Un interfaz, como el que se está discutiendo aquí, especifica un API (interfaz de programación de aplicación) para cierta funcionalidad. No define la implementación de dicho API. La sintaxis de la definición de un interfaz es normalmente semejante a la definición de una clase, excepto que no hay código definido para los métodos, sólo los nombres de los métodos, los argumentos necesarios y el tipo de valor devuelto. Un interfaz puede implementarse por una clase. Esto funciona de manera muy semejante a como lo hace la herencia. Si una clase implementa un interfaz, debe tener definidas en la clase las propiedades y métodos del interfaz. Los métodos que se implementan pueden codificarse de cualquier modo, siempre que el nombre, argumentos y tipo devuelto de cada método del interfaz sean idénticos a los que define el interfaz.

AGREGACIÓN

Las instancias de objetos que contienen otros objetos se denominan **objetos agregados**. La agregación puede conseguirse incluyendo el puntero a un objeto como valor en otro objeto. La ventaja de los objetos agregados es que permite la representación de estructuras complejas. Por ejemplo, un objeto contenido en un objeto agregado puede ser a su vez un objeto agregado.

Normalmente, las estructuras construidas con objetos agregados se limitan a una topología de árbol; esto es, no se permiten referencias circulares y cada instancia de objeto «hija» sólo puede tener una instancia de objeto «padre».

Es importante aclarar la diferencia entre la jerarquía de herencia de clases de objetos y la jerarquía de agregación de instancias de objeto. Las dos no están relacionadas. El uso de la herencia simplemente permite definir muchos tipos de objetos diferentes con un esfuerzo mínimo. El uso de la agregación permite la construcción de estructuras de datos complejas.

B.3. BENEFICIOS DEL DISEÑO ORIENTADO A OBJETOS

[CAST92] enumera los siguientes beneficios del diseño orientado a objetos:

- **Mejor organización de la complejidad inherente.** Mediante el uso de la herencia, pueden definirse eficientemente conceptos relacionados, recursos y otros objetos. Mediante el uso de la agregación pueden construirse estructuras de datos arbitrarias, que reflejen la tarea subyacente que se tiene entre manos. Los lenguajes de programación y las estructuras de datos orientadas a objetos permiten a los diseñadores describir recursos y funciones del sistema operativo de manera que reflejen el entendimiento del diseñador sobre dichos recursos y funciones.
- **Menor esfuerzo de desarrollo gracias a la reutilización.** La reutilización de clases de objetos que se han escrito, probado y mantenido por otros acorta los tiempos de desarrollo, prueba y mantenimiento.
- **Sistemas más extensibles y mantenibles.** El mantenimiento, incluyendo las mejoras y las correcciones del producto, tradicionalmente consume cerca del 65% del coste en la vida de cualquier producto. El diseño orientado a objetos reduce el porcentaje. La utilización de software

basado en objetos ayuda a limitar el número de interacciones potenciales entre diferentes partes del software, asegurando que pueden realizarse cambios en la implementación de una clase con muy poco impacto sobre el resto del sistema.

Estos beneficios están dirigiendo el diseño del sistema operativo en la dirección de los sistemas orientados a objetos. Los objetos permiten que los programadores ajusten un sistema operativo para satisfacer nuevos requisitos, sin comprometer la integridad del sistema. Los objetos también allanan el camino hacia la computación distribuida. Dado que los objetos se comunican mediante mensajes, no importa si los objetos están en el mismo sistema o en sistemas diferentes en una red. Datos, funciones e hilos pueden asignarse dinámicamente a estaciones de trabajo y servidores según se necesiten. Por consiguiente, el enfoque orientado a objetos para el diseño de sistemas operativos es cada vez más evidente en sistemas operativos para estaciones de trabajo y PC.

B.4. CORBA

Como hemos visto en este libro, los conceptos de orientación a objetos se han utilizado para diseñar e implementar núcleos de sistemas operativos, aportando beneficios en flexibilidad, gestionabilidad y portabilidad. Los beneficios de utilizar técnicas de orientación a objetos se extienden con igual o mayor beneficio al dominio del software distribuido, incluyendo los sistemas operativos distribuidos. La aplicación de técnicas de orientación a objetos en el diseño e implementación de software distribuido se conoce como computación de objetos distribuidos (*distributed object computing*, DOC).

La motivación para la DOC es la creciente dificultad de escribir software distribuido: a medida que el hardware de computación y de red se hacen más pequeños, más rápidos y más baratos, el software distribuido se hace mayor, más lento y más caro de desarrollar y mantener. [SCHM97] apunta que el reto del software distribuido surge de dos tipos de complejidad:

- **Inherente.** Las complejidades inherentes surgen de los problemas fundamentales de la distribución. Las principales son detectar y recuperar fallos en la red o en los nodos, minimizar el impacto de la latencia de las comunicaciones y determinar un particionamiento óptimo de componentes de servicio y de carga de trabajo sobre los computadores de una red. Además, la programación concurrente, con las cuestiones de bloqueo de recursos e interbloqueos, es aún difícil, y los sistemas distribuidos son inherentemente concurrentes.
- **Accidental.** Las complejidades accidentales surgen de las limitaciones de las herramientas y técnicas utilizadas para construir software distribuido. Un origen usual de la complejidad accidental es la amplia utilización del diseño funcional, que termina en sistemas no extensibles y no reutilizables.

DOC es un enfoque prometedor para la gestión de ambos tipos de complejidad. La pieza central de la propuesta DOC son los mediadores o *brokers* de solicitudes de objeto (*object request brokers*, ORB), que actúan como intermediarios en la comunicación entre objetos locales y remotos. Los ORB eliminan algunos de los aspectos tediosos, propensos a error y no portables del diseño e implementación de aplicaciones distribuidas. Complementando al ORB debe haber varias convenciones y formatos para el intercambio de mensajes, y definiciones del interfaz entre las aplicaciones y la infraestructura orientada a objetos

Hay tres tecnologías principales compitiendo en el mercado DOC: la arquitectura de OMG (*object management group*), llamada CORBA Arquitectura Común de Mediadores de Solicitudes de Objeto (*Common Object Request Broker Architecture*, CORBA); el sistema de invocación de métodos remotos de Java (*remote method invocation*, RMI); y el modelo distribuido de componentes de objetos de Microsoft (*Distributed Component Object Model*, DCOM). CORBA es el más avanzado y me-

jor asentado de los tres. Varios líderes de la industria, incluyendo IBM, Sun, Netscape y Oracle, soportan CORBA, y Microsoft ha anunciado que enlazará su esquema propietario DCOM con CORBA. El resto de este apéndice proporciona una breve visión general de CORBA.

La Tabla B.2 define algunos términos clave utilizados en CORBA. Las principales características de CORBA son (Figura B.4):

- **Clientes.** Los clientes generan solicitudes y acceden a servicios de objetos a través de variedad de mecanismos proporcionados por el ORB subyacente.
- **Implementaciones de objeto.** Estas implementaciones proporcionan los servicios solicitados por varios clientes del sistema distribuido. Un beneficio de la arquitectura CORBA es que ambas implementaciones de clientes y de objetos pueden ser escritas en cualquier lenguaje de programación y aún así proporcionar el conjunto completo de los servicios requeridos.
- **Núcleo ORB.** El núcleo ORB es el responsable de la comunicación entre objetos. El ORB encuentra un objeto en la red, entrega solicitudes al objeto, activa el objeto (si no está activo todavía) y devuelve cualquier mensaje de regreso al emisor. El núcleo ORB proporciona **transparencia de acceso** porque los programadores utilizan exactamente el mismo método y los mismos parámetros cuando invocan a un método local o a un método remoto. El núcleo ORB también proporciona **transparencia de localización**. Los programadores no necesitan especificar la localización de un objeto.
- **Interfaz.** Un interfaz de objeto especifica las operaciones y tipos soportados por el objeto y así define las solicitudes que se le pueden hacer al objeto. Los interfaces CORBA son similares a clases de C++ y a los interfaces de Java. A diferencia de las clases C++, un interfaz CORBA especifica métodos y sus parámetros y sus valores de retorno, pero no dice nada acerca de su implementación. Dos objetos de la misma clase C++ tienen la misma implementación de sus métodos.

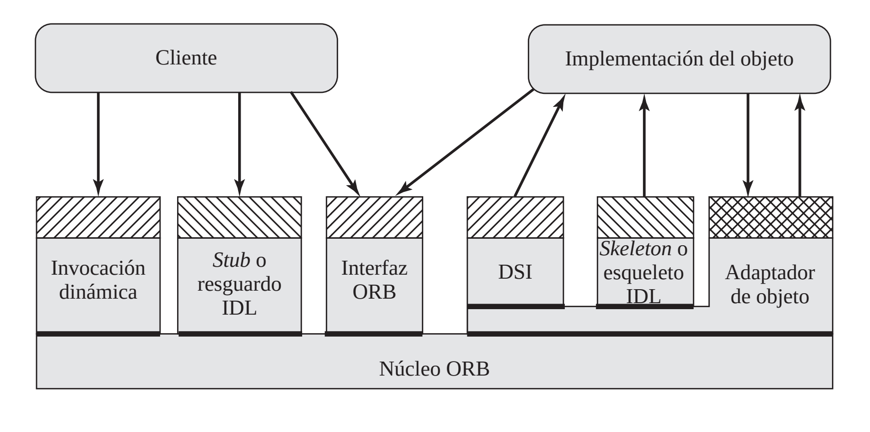


Figura B.4. Arquitectura CORBA (*Common Object Request Broker Architecture*).

Tabla B.2. Conceptos Clave en un Sistema Distribuido CORBA.

Concepto CORBA	Definición
Aplicación cliente	Invoca solicitudes a servidores para realizar operaciones sobre objetos. Una aplicación cliente utiliza una o más definiciones de interfaces que describen los objetos y operaciones que el cliente puede solicitar. Una aplicación cliente utiliza referencias a objetos, no objetos, para realizar las solicitudes.
Excepción	Contiene información que indica cuándo una solicitud se ha realizado satisfactoriamente.
Implementación	Define y contiene uno o más métodos que realizan el trabajo asociado con una operación de un objeto. Un servidor puede tener una o más implementaciones
Interfaz	Describe cómo se comportarán las instancias de un objeto, tal que qué operaciones son válidas sobre los objetos.
Definición de interfaz	Describe las operaciones disponibles sobre un tipo de objeto dado.
Invocación	El proceso de enviar una solicitud.
Método	El código de servidor que realiza el trabajo asociado a una operación. Los métodos están contenidos dentro de implementaciones.
Objeto	Representa una persona, sitio, cosa o pieza software. Sobre un objeto pueden realizarse operaciones, como la operación de promocionar a un objeto empleado.
Instancia de objeto	Una ocurrencia de un tipo de objeto particular.
Referencia a objeto	Un identificador de una instancia de objeto.
Lenguaje de definición de interfaces (IDL) de OMG	Un lenguaje de definición para la definición de interfaces en CORBA.
Operación	La acción que un cliente puede solicitarle a un servidor para que la realice sobre una instancia de objeto.
Solicitud	Un mensaje enviado de una aplicación cliente a una aplicación servidor.
Aplicación servidora	Contiene una o más implementaciones de objetos y sus operaciones.

- **Lenguaje de definición de interfaces de OMG (IDL).** IDL es el lenguaje utilizado para definir objetos. Un ejemplo de definición de interfaz en IDL es

```
//OMG IDL
interface Fábrica
    { Objeto crear();
};
```

Esta definición especifica un interfaz llamado Fábrica[1] que soporta una operación, crear. La operación crear no recibe parámetros y devuelve una referencia a un objeto de tipo Objeto.

---

[1] N. Del T. Fábrica, en inglés *Factory* es un interfaz común en el desarrollo de aplicaciones distribuidas, su esquema de aplicación es común en el vocabulario del desarrollo con patrones de diseño (*Design Patterns*). Es habitual, incluso en la literatura española que dicho término aparezca en inglés (*Factory*).

Dada una referencia a un objeto de tipo Fábrica, un cliente podría invocarlo para crear un nuevo objeto CORBA. IDL es un lenguaje independiente de la programación y, por esta razón, un cliente no invoca directamente ninguna operación del objeto. Para poder hacerlo se necesita una proyección del mismo en un lenguaje de programación del cliente. También es posible que servidor y cliente se programen en diferentes lenguajes. El uso de un lenguaje de especificación es una manera de abordar el procesamiento heterogéneo sobre entornos de múltiples lenguajes y plataformas. Así, IDL consigue **independencia de la plataforma**.

- **Creación de vínculos de lenguaje.** Los compiladores IDL convierten un archivo IDL de OMG a diferentes lenguajes de programación, que pueden ser o no orientados a objetos, tales como Java, Smalltalk, Ada, C, C++ o COBOL. Esta proyección incluye definiciones de los tipos de datos e interfaces de procedimientos específicos de lenguaje para acceder a objetos de servicio, el interfaz del resguardo del cliente IDL, el esqueleto IDL, los adaptadores de objeto, el interfaz de esqueleto dinámico y el interfaz directo de ORB. Normalmente, los clientes conocen en tiempo de compilación el interfaz del objeto y utilizan resguardos para realizar una invocación estática; en ciertos casos no se tiene tal conocimiento y deben hacer una invocación dinámica.
- **Resguardo IDL.** Realiza llamadas al núcleo ORB en nombre de una aplicación cliente. Los resguardos IDL proporcionan un conjunto de mecanismos que abstraen las funciones del núcleo de ORB en forma de mecanismos RPC (llamada a procedimiento remoto) que pueden emplearse por las aplicaciones cliente finales. Estos resguardos hacen que la implementación combinada de ORB y del objeto remoto parezca que estuviese enlazada dentro del mismo proceso. En la mayoría de los casos, los compiladores IDL generan bibliotecas de interfaces específicos de los lenguajes que completan el interfaz entre las implementaciones del cliente y del objeto.
- **Esqueleto IDL.** Proporciona el código que invoca métodos específicos del servidor. Los esqueletos IDL estáticos del lado del servidor complementan a los sustitutos IDL del lado del cliente. Incluyen los vínculos entre el núcleo ORB y las implementaciones de objeto que completan la conexión entre el cliente y las implementaciones de objeto.
- **Invocación dinámica.** Utilizando el interfaz de invocación dinámica (*dynamic invocation interface*, DII), una aplicación cliente puede invocar solicitudes sobre cualquier objeto sin tener que conocer en tiempo de compilación los interfaces del objeto. Los detalles del interfaz se rellenan consultando en un repositorio de interfaces o en otras fuentes en tiempo de ejecución. El DII permite a un cliente emitir mandatos de un único sentido (de los cuales no hay respuesta).
- **Interfaz de esqueleto dinámico (*Dynamic Skeleton Intereface*, DSI).** Parecido a la relación entre resguardos IDL y esqueletos IDL estáticos, el DSI ofrece entrega dinámica a objetos. Equivale a la invocación dinámica en el lado del servidor.
- **Adaptador de objetos.** Un adaptador de objetos es un componente del sistema CORBA proporcionado por un vendedor de CORBA para manejar tareas generales relacionadas con el ORB, tales como la activación de objetos y la activación de implementaciones. El adaptador toma estas tareas generales y las asocia a implementaciones y métodos particulares en el servidor.

B.5. LECTURAS Y PÁGINAS WEB RECOMENDADAS

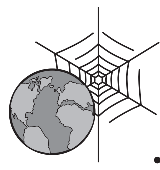
[KORS90] es una buena visión general de los conceptos de orientación a objetos. [STRO88] es una clara descripción de la programación orientada a objetos. [SYND93] proporciona una interesante perspectiva de los conceptos de orientación objetos. [VINO97] es una visión general de CORBA.

KORS90 Korson, T., and McGregor, J. «Understanding Object-Oriented: A Unifying Paradigm.» *Communications of the ACM*, Septiembre 1990.

STRO88 Stroustrup, B. «What is Object-Oriented Programming?» *IEEE Software*, Mayo 1988.

SNYD93 Snyder, A. «The Essence of Objects: Concepts and Terms.» *IEEE Software*, Enero 1993.

VINO97 Vinoski, S. «CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.» *IEEE Communications Magazine*, Febrero 1997.



PÁGINAS WEB RECOMENDADAS:

- ***Object Management Group.*** Consorcio industrial que promueve CORBA y otras tecnologías de objetos relacionadas.

APÉNDICE C

Proyectos de programación y de sistemas operativos

◆ ◆

Muchos profesores creen que los proyectos de implementación o investigación son cruciales para una compresión clara de los conceptos del sistema operativo. Sin proyectos, puede ser difícil para los estudiantes captar algunas de las abstracciones, así como interacciones entre componentes básicas del sistema operativo; un buen ejemplo de un concepto que muchos estudiantes encuentra difícil de dominar es el de los semáforos. Los proyectos refuerzan los conceptos presentados en este libro y dan al estudiante una mayor apreciación de cómo encajan entre sí las diferentes partes de un S.O. Asimismo, pueden motivar a los estudiantes y darles la confianza de que no sólo son capaces de entender los detalles de un S.O., sino también de implementarlos.

En este libro, se han intentado presentar los conceptos relacionados con los aspectos internos de los SS.OO. de una manera tan clara como sea posible y se han proporcionado numerosos ejercicios para reforzar estos conceptos. Sin embargo, muchos profesores desearán complementar este material con proyectos. Este apéndice proporciona una guía en este aspecto y describe material de apoyo disponible en el sitio web del profesor. En los Apéndices D y E se proporcionan detalles adicionales.

◆ ◆

C.1. PROYECTOS PARA LA ENSEÑANZA DE SISTEMAS OPERATIVOS

El profesor puede escoger entre las siguientes alternativas:

- **Proyectos de Sistemas Operativos (*Operating Systems Projects*, OSP).** OSP es tanto una implementación de un sistema operativo moderno como un entorno flexible para generar proyectos de implementación apropiados para un curso de introducción al diseño de SS.OO. OSP incluye varias propuestas de proyectos.
- **Intérprete Concurrente de Ben-Ari (*Ben-Ari Concurrent Interpreter*, BACI).** El BACI simula la ejecución de procesos concurrentes y proporciona semáforos binarios y con contador, así como monitores. BACI incluye varias propuestas de proyecto utilizadas para reforzar los conceptos de concurrencia.
- **Nachos.** Como OSP, Nachos es un entorno para generar proyectos de implementación para reforzar conceptos, incluyendo también diversas propuestas de proyectos.
- **Proyectos de investigación.** El sitio web del profesor propone una serie de tareas de investigación que se pueden asignar al estudiante para que éste estudie un determinado tema en Internet y escriba un informe sobre el mismo.
- **Proyectos de programación.** El sitio web del profesor proporciona un conjunto de pequeños proyectos de programación que pueden asignarse para reforzar los conceptos del libro, pudiéndose utilizar cualquier lenguaje. Los proyectos cubren un amplio rango de los temas tratados en este libro.
- **Tareas de lectura y análisis.** El sitio web del profesor incluye un lista de artículos importantes, uno o más por cada capítulo, que se pueden asignar a los estudiantes para que realicen un breve informe analizando el artículo.

Este apéndice proporciona un breve estudio de estos temas. El apéndice D proporciona una introducción más detallada de OSP, con información sobre cómo obtener el sistema y las propuestas de proyectos de programación. El apéndice E proporciona el mismo tipo de información para BACI. Nachos está bien documentado en su sitio web y se describe brevemente en la próxima sección.

C.2. NACHOS

INTRODUCCIÓN A NACHOS

Nachos es un sistema operativo pedagógico, que ejecuta como un proceso UNIX, para proporcionar a los estudiantes un entorno de depuración reproducible, que simule un sistema operativo y su hardware subyacente [CHRI93]. El objetivo de Nachos es proporcionar un entorno de proyectos que sea lo suficientemente realista para mostrar cómo funcionan los sistemas operativos reales, aunque suficientemente simple para que los estudiantes puedan comprenderlo y modificarlo de manera significativa.

En la Web está disponible un paquete de libre distribución que incluye:

- Un artículo de introducción.
- Un código inicial sencillo que corresponde con un sistema operativo en funcionamiento.
- Un simulador de un computador personal/estación de trabajo genérico.
- Propuestas de proyectos. Los proyectos enseñan y exploran todas las áreas de los sistemas operativos modernos, incluyendo hilos y concurrencia, multiprogramación, llamadas al sistema, memoria virtual, TLB gestionadas por software, sistemas de ficheros, protocolos de red, llamadas a procedimientos remotos y sistemas distribuidos.
- Una introducción a C++ (Nachos está escrito en un subconjunto de C++ de fácil aprendizaje, y esta introducción ayuda a enseñar este subconjunto a los programadores en C).

Nachos se utiliza en cientos de universidades a lo largo del mundo y se ha adaptado a numerosos sistemas, incluyendo Linux, FreeBSD, NetBSD, DEC MIPS, DEC Alpha, Sun Solaris, SGI IRIS, HP-UX, IBM AIX, MS-DOS y Apple Macintosh. Los planes futuros incluyen una adaptación al SimOS de Stanford, una simulación completa de una estación de trabajo SGI.

Nachos está disponible de manera gratuita en su sitio Web (hay un enlace a su sitio web en WilliamStallings.com/OS/OS5e.html); está disponible un conjunto de soluciones para los profesores mediante correo electrónico a nachos@cs.berkeley.edu. Además, hay una lista de correo para profesores y un grupo de noticias (alt.os.nachos).

CÓMO ELEGIR ENTRE NACHOS, OSP Y BACI

Si el profesor está dispuesto a dedicar un cierto tiempo a adaptar uno de estos tres simuladores al entorno local disponible para los estudiantes, la elección entre los tres dependerá de los objetivos y la opinión personal del profesor. Si el enfoque del proyecto es la concurrencia, BACI es la elección adecuada. BACI proporciona un entorno excelente para el estudio de las complejidades y la sutilezas de los semáforos, los monitores y la programación concurrente.

En cambio, si el profesor desea que los estudiantes exploren diversos mecanismos del S.O., incluyendo programación concurrente, espacios de direcciones y planificación, memoria virtual, sistemas de ficheros, redes, etcétera, entonces se puede utilizar Nachos u OSP.

En este libro se ha incluido un apéndice sobre OSP debido a que es uno de los mejores vehículos disponibles para dar soporte a proyectos de SS.OO. OSP se utiliza en más de 100 sitios y proporciona una gran cantidad de apoyo y documentación. Un inconveniente potencial es que, aunque el sistema, los proyectos propuestos y la lista de correo asociada son gratuitos, hay un pequeño manual de usuario que los estudiantes deberían comprar. Sin embargo, esto se debe compensar con los puntos fuertes

de este entorno. De manera similar, Nachos es un sistema muy utilizado y proporciona soporte, documentación y propuestas de proyectos. Se anima al profesor a que estudie el apéndice D y, si está interesado, lo compare con el artículo de introducción a Nachos y otra documentación disponible en el sitio web de Nachos.

C.3. PROYECTOS DE INVESTIGACIÓN

Una manera efectiva de reforzar los conceptos básicos del curso y enseñar a los estudiantes técnicas de investigación es asignarles un proyecto de investigación. Este tipo de proyecto podría involucrar una búsqueda en la bibliografía, así como en la Web, de los productos comerciales, las actividades de investigación de laboratorio y los esfuerzos de estandarización. Los proyectos pueden asignarse a equipos o, en el caso de proyectos más pequeños, de forma individual. En cualquier caso, es mejor solicitar al alumno algún tipo de propuesta de proyecto al principio del curso, dando de esta manera al profesor tiempo para evaluar que la propuesta tenga el tema y el nivel de dificultad apropiados. La documentación entregada a los estudiantes para los proyectos de investigación debería incluir:

- Un formato para la propuesta.
- Un formato para el informe final.
- Una planificación con plazos intermedios y finales.
- Una lista de posibles temas para proyectos.

Los estudiantes pueden seleccionar uno de los temas de la lista o diseñar su propio proyecto equiparable. El sitio web del profesor incluye un formato sugerido para la propuesta y el informe final, además de una lista de posibles temas de investigación desarrollada por el profesor Tan N. Nguyen de la Universidad George Mason.

C.4. PROYECTOS DE PROGRAMACIÓN

Una alternativa al desarrollo de fragmentos de un SO, utilizando OSP o Nachos, o centrándose en la concurrencia, utilizando BACI, es asignar varios proyectos de programación que no requieren infraestructura. Hay varias ventajas en los proyectos de programación frente al uso de un entorno de trabajo de apoyo como OSP o BACI:

1. El profesor puede elegir entre una gran variedad de conceptos relacionados con el SO para asignar proyectos, no sólo aquéllos que encajen en el entorno de trabajo de apoyo.
2. Los estudiantes pueden programar el proyecto en cualquier computador disponible y en cualquier lenguaje apropiado: son independientes de la plataforma y del lenguaje.
3. El profesor no necesita descargar, instalar ni configurar la infraestructura.

Hay también flexibilidad en el tamaño de los proyectos. Los proyectos más grandes proporcionan a los estudiantes una mayor sensación de éxito, pero los estudiantes con menos capacidad o con menos habilidades organizativas pueden retrasarse. Los proyectos más grandes normalmente proporcionan un mayor beneficio global a los mejores estudiantes. Los proyectos más pequeños pueden tener una mayor proporción entre conceptos y código y, dado que se puede asignar un mayor número de ellos, existe la oportunidad de tratar varios temas diferentes. Globalmente, la ventaja parece inclinarse hacia los proyectos más pequeños. Por tanto, el sitio web del profesor contiene varios pequeños proyectos, cada uno destinado a completarse en aproximadamente una semana, lo cual puede ser muy

satisfactorio tanto para el estudiante como para el profesor. Estos proyectos los desarrolló Stephen Taylor en el Instituto Politécnico de Worcester, quien ha utilizado y mejorado los proyectos del curso de sistemas operativos una docena de veces.

Además, en el libro se bosquejan otros dos proyectos de programación de cierta envergadura, al final de los Capítulos 3 y 9. En el sitio web del profesor se proporciona un conjunto gradual y más detallado de instrucciones para estos dos proyectos.

C.5. TAREAS DE LECTURA Y DE ANÁLISIS

Otra excelente manera de reforzar los conceptos del curso y proporcionar a los estudiantes experiencia en investigación es asignarles artículos de la bibliografía del tema para que los lean y analicen. El sitio web del profesor incluye una propuesta de artículos para asignar a los estudiantes, organizados por capítulos. Todos los artículos están disponibles en Internet o en cualquier buena biblioteca técnica de una universidad. El sitio web del profesor también incluye un documento con ejercicios propuestos.

APÉNDICE D

OSP. Un entorno para proyectos de sistemas operativos*

* Por Michael Kifer and Scott A. Smolka, Departamento de Informática SUNY en Stony Brook, {kifer, sas}@cs.sunysb.edu.

D.1. INTRODUCCIÓN

OSP2 y OSP son ambos una implementación de un sistema operativo moderno y un entorno flexible para generar proyectos de implementación apropiados para un curso introductorio de diseño de sistemas operativos [KEFE92]. Tiene como objetivo complementar el uso de un libro de texto introductorio de sistemas operativos y contiene suficientes proyectos para tres semestres. Estos proyectos exponen a los estudiantes muchas características esenciales de los sistemas operativos y a la vez los aislan de muchos aspectos de bajo nivel dependientes de máquina. Por tanto, incluso en un semestre, los estudiantes pueden aprender estrategias de reemplazo de páginas en la gestión de la memoria virtual, estrategias de planificación del procesador, optimizaciones del tiempo de búsqueda en disco y otros aspectos relacionados con el diseño de los sistemas operativos. Al mismo tiempo, ambos sistemas proporcionan entornos adecuados en los cuales crear y administrar proyectos de implementación para los estudiantes, automatizando este trabajo rutinario para el profesor. Los proyectos se pueden organizar en cualquier orden deseado de forma que progresen de una manera consistente con el material de las clases.

Los estudiantes programan sus proyectos OSP 2 en el lenguaje de programación Java. Adicionalmente, el sistema original OSP, que se basa en el lenguaje de programación C, también se encuentra disponible. El nuevo sistema OSP 2 se puede utilizar tanto en un entorno Windows como en un entorno UNIX/Linux; el sistema OSP original trabaja sólo sobre UNIX/Linux. Mientras que el modelo del sistema operativo subyacente en OSP 2 es más moderno y tiene más características, algunos profesores prefieren que sus estudiantes programen en C. Esta necesidad es satisfecha por el sistema OSP original.

Tanto OSP2 como OSP están compuestos por varios módulos, cada uno de los cuales lleva a cabo un servicio de un sistema operativo básico, tal como planificación de dispositivos, planificación de procesador, manejo de interrupciones, gestión de ficheros, gestión de memoria, gestión de procesos, gestión de recursos y comunicación entre procesos. Omitiendo selectivamente cualquier subconjunto de módulos, el profesor puede generar un proyecto en el cual los estudiantes tengan que implementar las partes restantes. Este proceso está completamente automatizado. Los proyectos se pueden organizar en cualquier orden deseado de forma que progresen de una manera consistente con el material de las clases.

Cada proyecto está formado por un «módulo de carga parcial» de módulos estándar a los cuales los estudiantes enlazan sus implementaciones de los módulos asignados. El resultado es un sistema operativo nuevo y completo, parcialmente implementado por los estudiantes. Los proyectos también traen plantillas de módulos, que contienen declaraciones de las estructuras de datos requeridas por cada uno de los módulos asignados. Estos ficheros son parte de una asignación de un proyecto en los cuales los estudiantes tienen que rellenar los cuerpos de los procedimientos. De esta forma, los ficheros de plantilla ayudan a eliminar gran parte de la rutina de escritura del código, tanto para el profesor como para los estudiantes.

Los núcleos de OSP2 y OSP son simuladores que ofrecen la ilusión de un sistema de computación con una colección dinámica de procesos de usuario que deben multiprogramarse. El resto de los módulos de OSP2 y OSP se construyen para responder apropiadamente a los eventos generados por el simulador y que guían al sistema operativo. El simulador «comprende» su interacción con otros módulos y puede frecuentemente detectar una respuesta errónea por parte de un módulo a un evento simulado. En dichos casos, el simulador terminará de forma ordenada la ejecución del programa, enviando un mensaje de error significativo al usuario, indicando dónde se puede encontrar el error. Esta utilidad sirve tanto como una herramienta de depuración para el estudiante como una herramienta docente para el profesor, que asegura que los programas del estudiante están prácticamente libres de errores.

Se puede ajustar dinámicamente la dificultad de los flujos de trabajo generados por el simulador a través de la manipulación de los parámetros de simulación. Esto lleva a una forma sencilla y efectiva de comprobar la calidad de los programas de los estudiantes. Hay también utilidades que permiten a los estudiantes depurar sus programas interaccionando con OSP2 u OSP durante la simulación.

El modelo subyacente de OSP2 y OSP no es un clon de un sistema operativo específico. Por el contrario, es una abstracción de las características encontradas comúnmente en varios sistemas (aunque en ocasiones se puede ver una tendencia hacia UNIX). Aunque los módulos de OSP2 y OSP se diseñaron para esconder muchos de los aspectos de bajo nivel que se pueden encontrar en el diseño e implementación de los sistemas operativos, los módulos tratan los aspectos más importantes de los sistemas modernos de la vida real. Su implementación se adapta adecuadamente al componente de proyectos de un curso introductorio de sistemas operativos. OSP2 se describe en el siguiente libro, publicado por Addison-Wesley a finales de 2004:

> *Introduction to Operating System Design and Implementation: The OSP 2 Approach*, Michael Kifer y Scott A. Smolka, Addison-Wesley.

El OSP original se documenta en el siguiente libro, que está todavía disponible en la editorial:

> *OSP: An Environment for Operating System Projects.* Michael Kifer y Scott A. Smolka, Addison-Wesley, ISBN 0-201-54887-9 (1991).

En el sitio web de OSP se encuentran disponibles manuales introductorios sólo para profesores: http://www.cs.sunysb.edu/osp. Los usuarios se deben registrar en el sitio y se debe verificar que son profesores.

D.2. ASPECTOS INNOVADORES DE OSP

Los principales aspectos innovadores de OSP 2 y OSP son los siguientes:

- Los módulos que los estudiantes escriben a lo largo del curso de un semestre se construyen para responder apropiadamente a los eventos generados por el simulador que dirige al sistema operativo. Los estudiantes pueden ajustar la dificultad de los flujos de trabajos generados por el simulador mediante valores diferentes para los parámetros de simulación requeridos. Esto conduce a una forma sencilla y efectiva para los estudiantes de comprobar la calidad de sus programas. Hay también utilidades que permiten a los estudiantes depurar sus programas interaccionando con OSP2 u OSP durante la simulación.
- OSP proporciona al profesor un entorno adecuado, el generador de proyectos OSP, en el cual crear proyectos de implementación. El generador de proyectos genera un «módulo de carga parcial» de módulos OSP estándar a los cuales los estudiantes tienen que enlazar sus implementaciones de los módulos asignados. El resultado es un sistema operativo nuevo y completo, parcialmente implementado por el estudiante. OSP 2 también proporciona un generador de proyectos pero viene equipado adicionalmente con proyectos preensamblados; esto minimiza el esfuerzo del profesor requerido para desarrollar los proyectos de los estudiantes.

  Adicionalmente, el generador de proyectos crea automáticamente los ficheros de plantilla que contienen cabeceras de procedimientos y declaraciones de estructuras de datos requeridas por cada uno de los módulos asignados. Estos ficheros son parte de la asignación de un proyecto en el que los estudiantes tienen que rellenar los cuerpos de los procedimientos. Esto asegura una interfaz consistente a OSP y elimina gran parte de la rutina de escritura del código, tanto para el profesor como para los estudiantes.

- OSP 2 y OSP también incluyen sistemas de envío de proyectos opcionales, que los profesores pueden utilizar para que sus estudiantes envíen sus trabajos de una forma conveniente y «segura». El sistema de entrega compilará los ficheros fuente de los estudiantes y ejecutará los ejecutables con un parámetro dado.

  Los ficheros fuente de los módulos implementados por los estudiantes, la salida de la fase de compilación y las ejecuciones de la simulación se colocan en un directorio de entregas en la cuenta del curso. Los estudiantes no tienen acceso a estos ficheros de salida, de forma que no puedan alterarse maliciosamente.

  Los estudiantes pueden entregar sus programas cualquier número de veces. Cada nueva entrega sobrescribe las anteriores. Cada entrega tiene asociada un sello de tiempo, de forma que el profesor pueda verificar los plazos de entrega.

- Los simuladores OSP 2 y OSP monitorizan cuidadosamente el comportamiento en tiempo real de los módulos implementados por los estudiantes y en muchos casos pueden mostrar un mensaje de advertencia cuando el comportamiento del módulo de un estudiante se desvíe de la norma. Por ejemplo, si el estudiante ha escrito el módulo que maneja las peticiones de E/S de los usuarios, el simulador comprobará que un bloque de peticiones de E/S (*I/O Request Block*, IORB) se ha insertado adecuadamente en la cola de dispositivos apropiada. Si no, el simulador enviará un mensaje descriptivo que avisará al estudiante de que la petición de E/S no ha sido manejada correctamente.

  La monitorización llevada a cabo por el simulador constituye una ayuda para la depuración interactiva y es valiosa para el estudiante: Casi siempre se cumple que si la ejecución de la solución de un estudiante se realiza sin ningún mensaje de advertencia, entonces el código del estudiante es funcionalmente correcto.

- OSP2 y OSP proporcionan una amplia interfaz de depuración al simulador, que permite que los estudiantes periódicamente vean (tomen una instantánea) el estado del sistema durante la simulación. La información mostrada durante una instantánea incluye los contenidos de la tabla de marcos de memoria principal, las tablas de dispositivos, el conjunto de BCP (Bloque de Control del Proceso) y las colas de eventos. Una instantánea también ofrece al usuario una oportunidad para cambiar los parámetros de simulación. Uno de dichos parámetros, el `snapshot_interval/`, podría modificarlo el usuario para indicar con qué frecuencia durante la simulación debe mostrarse el estado del sistema.

- OSP 2 y OSP están documentados completamente en dos libros de texto; estos libros proporcionan a los estudiantes toda la información que necesitan para completar sus trabajos, incluyendo especificaciones detalladas de cada uno de los módulos e instrucciones para compilar, ejecutar y entregar sus trabajos. El manual del profesor añade información sobre cómo instalar cada sistema e instrucciones para utilizar el sistema de entrega.

  El sitio web de OSP, http://www.cs.sunysb.edu/osp/, contiene varias prácticas de programación de OSP 2/OSP que han asignado diferentes profesores a sus estudiantes.

- El sitio web de OSP proporciona a los usuarios registrados un foro de discusión. El propósito del foro es servir como medio de discusión sobre OSP a sus usuarios y como una manera de informar a la comunidad OSP de cualquier cambio en el software y la documentación (por ejemplo, resoluciones de errores, mejoras, versiones futuras).

D.3. COMPARACIÓN CON OTRAS HERRAMIENTAS DOCENTES DE SISTEMAS OPERATIVOS

Las herramientas docentes de sistemas operativos se pueden clasificar en dos grupos: aquellas basadas en simulación (por ejemplo, el sistema operativo Toy de Berkeley y Nachos, MPX) y aquéllas ba-

sadas en el código fuente de sistemas operativos reales que ejecutan directamente en una máquina desnuda (por ejemplo, MINIX [TANE97], XINU [COME84] y Linux). OSP cae claramente en la primera categoría. Se puede considerar que las dos categorías de herramientas docentes suponen dos nichos diferentes en la educación de los sistemas operativos. El software sobre la máquina desnuda tiene como ventaja el hecho de que permite al estudiante conocer íntimamente los detalles de bajo nivel de la arquitectura de máquina y produce una sensación de inmediatez que no está presente en el software basado en simulación. Por otro lado, el software basado en simulación, libera intencionadamente al estudiante de los aspectos básicos de cualquier arquitectura particular de máquina y permite al estudiante centrarse en la implementación de los conceptos de sistemas operativos discutidos en clase o en el texto del curso.

Dentro de las diferentes herramientas basadas en simulación, OSP y OSP2 se distinguen por la siguiente combinación de atributos:

- **Flexibilidad.** Los profesores tienen completa libertad para asignar proyectos sobre sus temas favoritos, en cualquier orden. Más aún, cada proyecto no está atado a una estrategia de disco específica, a una planificación del procesador concreta, a una gestión de memoria determinada, a una política para evitar interbloqueos, u otros aspectos de diseño específicos.
- **Grado de realismo ofrecido por el simulador.** OSP2 y OSP se basan en una simulación creíble de eventos que se dan en un sistema operativo típico, de modo que el profesor está adecuadamente equipado para evaluar la calidad de las implementaciones de los estudiantes. El simulador tiene incorporada una serie de comprobaciones de seguridad que hacen difícil falsificar los resultados y por tanto simplifican la verificación de los proyectos de un estudiante.
- **Facilidad de uso.** La experiencia ha mostrado que OSP2 y OSP son relativamente fáciles de utilizar tanto desde la perspectiva del profesor como del estudiante. Al profesor se le quita la carga administrativa de asignar a mano los proyectos. Éstos los genera automáticamente el generador de proyectos y contienen toda la información necesaria para que el estudiante complete sus asignaciones. La interfaz de depuración para el simulador decrementa significativamente el tiempo que un estudiante necesita utilizar para completar una práctica. Finalmente, el sistema de entrega de proyectos hace la vida más fácil tanto al profesor como al estudiante.

Obsérvese que el sitio web de OSP mencionado anteriormente está dirigido exclusivamente para uso de profesores de cursos de sistema operativo que utilizan el software de OSP2 y OSP. En particular, no se encuentra orientado a estudiantes de dichos cursos. Por tanto, cada usuario registrado debe autenticarse como un profesor antes de acceder a ninguna información.

APÉNDICE E

BACI. El sistema de programación concurrente de Ben-Ari*†

* Por Hill Bynum, *College of William and Mary* y Tracy Camp, *Colorado Scholl of Mines*.

† Este trabajo fue financiado parcialmente por la *National Science Foundation* (NSF) NCR-9702449.

E.1. INTRODUCCIÓN

En el Capítulo 5, se introducen los conceptos de concurrencia (ej., exclusión mutua y el problema de la sección crítica) y se proponen técnicas de sincronización (ej., semáforos, monitores y paso de mensajes). Los aspectos de interbloqueo y hambruna en los programas concurrentes se exponen en el Capítulo 6. Dado el énfasis creciente en la computación paralela y distribuida, entender la concurrencia y la sincronización es más necesario que nunca. Para conseguir un entendimiento profundo de estos conceptos se necesita experiencia práctica escribiendo programas concurrentes.

Existen tres opciones para conseguir esta deseada experiencia directa. Primero, podemos escribir programas concurrentes con un lenguaje de programación concurrente conocido, como Pascal Concurrente, Modula, Ada o el lenguaje de programación SR. Sin embargo, para experimentar con diversas técnicas de sincronización debemos aprender la sintaxis de muchos lenguajes de programación concurrente. Segundo, podemos escribir programas concurrentes usando las llamadas al sistema de un sistema operativo como UNIX. Es fácil, sin embargo, que nos distraigamos del objetivo de entender la programación concurrente por los detalles y peculiaridades de un sistema operativo particular (ej., detalles de las llamadas al sistema para semáforos en UNIX). Por último, podemos escribir programas concurrentes con un lenguaje desarrollado específicamente para proporcionar experiencia en los conceptos de concurrencia como el intérprete concurrente Ben-Ari (BACI) [BYNU96]. El uso de un lenguaje como éste ofrece variedad de técnicas de sincronización con una sintaxis que es normalmente familiar. Los lenguajes desarrollados específicamente para proporcionar experiencia en los conceptos de concurrencia son la mejor opción para obtener la deseada experiencia directa.

La Sección E.2 contiene una breve visión general del sistema BACI y explica cómo obtener el sistema. La Sección E.3 contiene ejemplos de programas BACI, y la Sección E.4 contiene una exposición de proyectos para obtener experiencia práctica sobre concurrencia a los niveles de implementación y programación. Finalmente, la Sección E.5 contiene una descripción de los cambios del sistema BACI que están en progreso o han sido planeados.

E.2. BACI

VISIÓN GENERAL DEL SISTEMA

BACI es un descendiente directo de la modificación de Ben-Ari del Pascal secuencial (Pascal-S). Pascal-S es un subconjunto del Pascal de Wirth, sin ficheros, excepto INPUT y OUTPUT, ni conjuntos, variables puntero o sentencias *goto*. Ben-Ari tomó el lenguaje Pascal-S y añadió construcciones de programación concurrente tales como la construcción `cobegin . . . coend` y el tipo de variable semáforo con las operaciones `wait` y `signal` [BEN82]. BACI es la modificación de Pascal-S con nuevas características de sincronización (ej., monitores) así como mecanismos de encapsulación para asegurar que el usuario no puede modificar una variable de forma inapropiada (ej., una variable semáforo sólo puede ser modificada por las funciones del semáforo).

BACI simula la ejecución concurrente de procesos y da soporte a las siguientes técnicas de sincronización: semáforos generales, semáforos binarios y monitores. El sistema BACI se compone de dos subsistemas, como ilustra la Figura E.1. El primer subsistema, el compilador, compila el programa de usuario a un código objeto intermedio denominado PCODE. Hay dos compiladores disponibles en el sistema BACI, correspondientes a dos tipos populares de lenguajes de cursos de introducción a la programación. La sintaxis de uno de los compiladores es similar al Pascal estándar; los programas BACI que utilizan sintaxis Pascal se denotan como nombre-prog.pm. La sintaxis del otro compilador es similar al C++ estándar; estos programas BACI se denotan como nombre-prog.cm.

Ambos compiladores crean dos archivos durante la compilación: `nombre-prog.lst` y `nombre-prog.pco`.

El segundo subsistema del sistema BACI, el intérprete, ejecuta el código objeto creado por el compilador. En otras palabras, el intérprete ejecuta `nombre-prog.pco.` El núcleo del intérprete es un planificador expulsivo; durante la ejecución, este planificador alterna aleatoriamente los procesos concurrentes, simulando así la ejecución paralela de los procesos concurrentes. El intérprete ofrece varias opciones de depuración, tales como la ejecución paso a paso, desensamblar instrucciones de PCODE o mostrar posiciones de almacenamiento del programa.

CONSTRUCCIONES CONCURRENTES EN BACI

En el resto de este apéndice, nos centramos en el compilador de sintaxis similar al C++ estándar. Llamamos a este compilador C—; aunque la sintaxis es similar a la de C++, no incluye herencia, ni encapsulación ni otras características de programación orientada a objetos. En esta sección, damos una visión general de las construcciones concurrentes de BACI; acuda a las guías de usuario del sitio Web de BACI para obtener más detalles sobre la sintaxis tipo Pascal o C— de BACI.

cobegin. Una lista de procesos que deben ejecutar concurrentemente tiene que indicarse dentro de un bloque cobegin. Tales bloques no pueden anidarse y deben aparecer en el programa principal.

cobegin{ proc1(…); proc2(…); … ; procN(…); }

Las sentencias PCODE que crea el compilador para el bloque anterior serán entrelazadas por el intérprete en un orden arbitrario «aleatorio»; múltiples ejecuciones del mismo programa que contenga un bloque cobegin aparentarán ser no deterministas.

Semáforos. Un semáforo en BACI es una variable `int` de valor no negativo, que sólo puede ser accedida por las llamadas de semáforos definidas a continuación. Un semáforo binario en BACI, uno

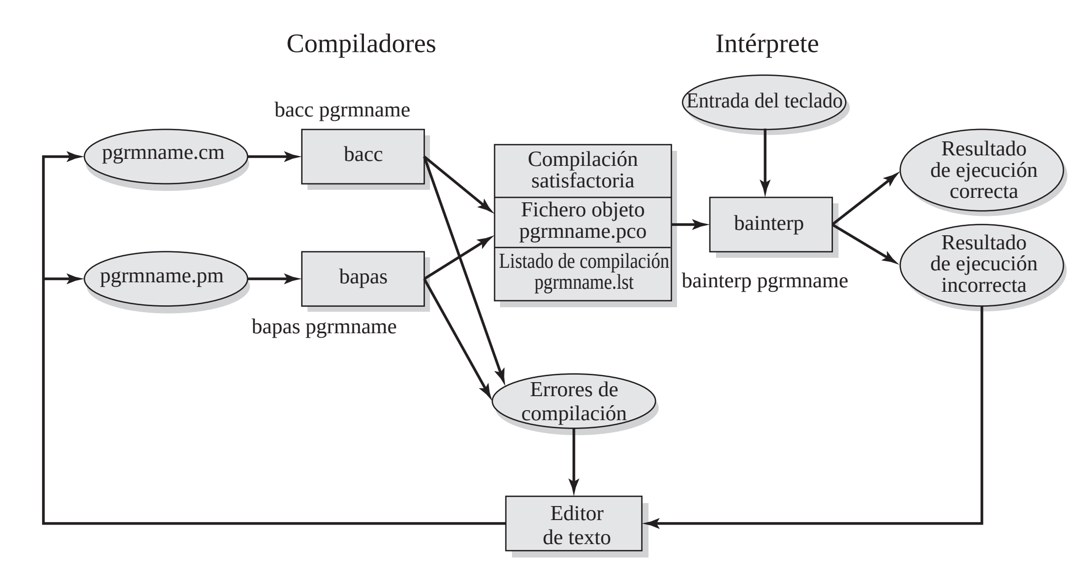


Figura E.1. Visión general del sistema BACI.

que sólo toma los valores 0 y 1, se define con el subtipo `binarysem` del tipo `semaphore`. Durante la compilación y ejecución, el compilador y el intérprete hacen cumplir las restricciones de que una variable `binarysem` sólo puede tomar los valores 0 o 1 y de que un tipo semáforo sólo puede ser no negativo. Las llamadas de semáforos de BACI incluyen:

- `initialsem(semaphore sem, in expresion)`
- `p(semaphore sem)`. Si el valor de `sem` es mayor de cero, el intérprete decrementa `sem` y retorna, permitiendo que quien llamó a `p` continúe. Si el valor de `sem` es igual a cero, el intérprete debe hacer que quien llamó a `p` duerma. El mandato `wait` se acepta como sinónimo de `p`.
- `v(semaphore sem)`. Si el valor de `sem` es igual a cero y uno o más procesos están durmiendo en `sem`, se despierta a uno de estos procesos. Si no hay procesos esperando en `sem`, se incrementa `sem` en uno. En cualquier caso a quien llama a `v` se le permite continuar. (BACI se ajusta a la propuesta de semáforos original de Dijkstra al escoger aleatoriamente el proceso a despertar cuando llega una señal). El mandato `signal` se acepta como sinónimo de `v`.

Monitores BACI da soporte al concepto de monitor tal como lo propuso Hoare [HOAR74] con ciertas restricciones; la implementación se basa en el trabajo realizado por [PRAM84]. Un monitor es un bloque C++, como el bloque definido por un procedimiento o función, con algunas propiedades adicionales (ejemplo, variables condición). En BACI, un monitor debe ser declarado en el nivel global más externo y no puede estar anidado dentro de otro bloque monitor. Las funciones y procedimientos de un monitor utilizan tres construcciones para controlar la concurrencia: variables condición, `waitc` (esperar en una condición) y `signalc` (señalar una condición). Una condición jamás «tiene» realmente un valor; es algo donde esperar o algo a señalar. Un proceso monitor puede esperar a que una condición suceda o señalar que una condición dada sucede ahora a través de las llamadas `waitc` y `signalc`. Estas llamadas tienen la siguiente sintaxis y semántica:

- `waitc(condition cond, int prio)`. El proceso monitor (y por tanto el proceso exterior que invoca al proceso monitor) se bloquea sobre la condición `cond` y se le asigna la prioridad `prio`.
- `waitc(condition cond)`. Esta llamada tiene la misma semántica que la llamada `waitc`, pero se le asigna una prioridad por defecto de 10.
- `signalc(condition cond)`. Despertar al proceso que esté esperando en `cond` que tenga la prioridad más alta (menor valor); si ningún proceso espera en `cond`, no hacer nada.

BACI cumple el requisito de reanudación inmediata. En otras palabras, si se señala a un proceso que está esperando en una condición, este tendrá prioridad sobre otros procesos que intenten entrar en el monitor.

Otras construcciones concurrentes. El compilador de C- - de BACI proporciona varias construcciones concurrentes de bajo nivel que pueden utilizarse para crear nuevas primitivas de control de concurrencia. Si una función se define como atómica, entonces tal función no es expulsable. En otras palabras, el intérprete no interrumpirá a una función atómica con un cambio de contexto. En BACI, la función *suspend* hace que el proceso que la invoca duerma y la función *revive* retoma un proceso suspendido.

CÓMO OBTENER BACI

El sistema BACI, con dos guías de usuario (una para cada uno de los dos compiladores) y descripciones detalladas de proyectos, está disponible en el sitio web de BACI (hay un enlace a su sitio web

desde WilliamStallings.com/OS/OS5e.html). El sistema BACI está escrito en C y por tanto es fácilmente portable. Por tanto, el sistema BACI puede ser compilado en Linux, RS/6000 AIX, Sun OS, DOS y CYGWIN sobre Windows con mínimas modificaciones sobre el fichero Makefile. (En el fichero README de la distribución se indican los detalles de instalación para una plataforma dada). Puede contactar con los autores en bynum@cs.wm.edu o tcamp@mines.edu.

E.3. EJEMPLOS DE PROGRAMAS BACI

En los Capítulos 5 y 6 se expusieron varios problemas clásicos de sincronización (ej., el problema de los lectores/escritores y el problema de los filósofos comensales). En esta sección se ilustra el sistema BACI con tres programas. El primer ejemplo ilustra la ejecución no determinista de procesos concurrentes en el sistema BACI. Considere el siguiente programa:

```
const int m = 5;
int n;
void incr(char id)
{
    int i;
    for(i = 1; i <= m; i = i + 1)
    {
        n = n + 1;
        cout << id << " n =" << n << " i =";
        cout << i << " " << id << endl;
    }
}
main( )
{
    n = 0;
    cobegin {
        incr('A'); incr('B' ); incr('C');
    }
    cout << "La suma es " << n << endl;
}
```

Observe en el programa precedente que si los tres procesos creados (A, B y C) se ejecutaran en secuencia, la suma producida sería 15. Sin embargo, la ejecución concurrente de la sentencia `n = n + 1;`, puede dar lugar a diferentes valores de suma. Tras compilar el programa precedente con bacc, se ejecutó varias veces el archivo PCODE con el bainterp. Cada ejecución produjo sumas entre 9 y 15. El siguiente es un ejemplo de la ejecución producida por el intérprete BACI:

```
Fichero fuente: incremen.cm Vie 1 Ago 16:51:00 1997
CB n =2 i =1 C n =2
A n =2 i =1 i =1 A
CB
    n =3 i =2 C
```

```
A n =4 i =2 C n =5 i =3 C
A
B n =6C i =2 B
    n =7 i =4 C
A n =8 i =3 A
BC n =10 n =10 i =5 C
A n = i =311 i =4 A
    B
A n =12 i =B5 n =13A
    i =4 B
B n =14 i =5 B
La suma es 14
```

Se necesitan instrucciones máquina especiales para sincronizar el acceso de los procesos a una memoria principal común. Luego, sobre estas instrucciones especiales, se construyen protocolos de exclusión mutua y primitivas de sincronización. En BACI, el intérprete no interrumpirá con un cambio de contexto a una función definida como atómica. Esta característica permite a los usuarios implementar estas instrucciones máquina especiales de bajo nivel. Por ejemplo, el siguiente programa es una implementación BACI de la función `testset` definida en la Sección 5.2.

```
// Instrución test and set
// Stallings, Sección 5.2
//
atomic int testset(int& i)
{
    if (i == 0) {
        i = 1;
        return 1;
    }
    else
        return 0;
}
```

Podemos usar `testset` para implementar protocolos de exclusión mutua, como se muestra en el siguiente programa. Este programa es una implementación BACI de un programa de exclusión mutua basado en la instrucción *test and set*. El programa lanza tres procesos concurrentes; cada proceso solicita exclusión mutua 10 veces.

```
int cerrojo = 0;
const int Repeticiones = 10;
void proc(int id)
{
    int i = 0;
    while(i < Repeticiones) {
```

```
            while (testset(cerrojo)); // esperar
                // entrar en sección crítica
                cout << id;
            // abandonar la sección crítica
            cerrojo = 0;
            i++;
        }
}
main( )
{
    cobegin {
        proc(0); proc(1); proc(2);
    }
}
```

Los siguientes dos programas son una solución BACI al problema productor/consumidor de *buffer* acotado con semáforos (*véase* la Figura 5.13). En este ejemplo, tenemos dos productores, tres consumidores y un tamaño de *buffer* de 5. Primero se enumeran los detalles del programa para este problema. Luego se incluyen los archivos que definen la implementación del *buffer* acotado.

```
// Una solución al problema productor/consumidor de buffer acotado.
// Stallings, Figura 5.13
// Introducir la maquinaria del buffer acotado.
#include «boundedbuff.inc»
const int RangoValores = 20; // se producirán enteros en el rango 0..19
semaphore ts; // para acceso exclusivo al terminal de salida
semaphore s; // exclusión mutua sobre el buffer
semaphore n; // número de elementos consumibles en el buffer
semaphore e; // número de espacios vacíos en el buffer
int producir(char id)
{
    int tmp;
    tmp = random(RangoValores);
    wait(ts);
    cout << "Productor " << id << " produce " << tmp << endl;
    signal(ts);
    return tmp;
}
void consumir(char id, int i)
{
    wait(ts);
    cout << "Consumidor " << id << " consume " << i << endl;
    signal(ts);
```

```
}
void productor(char id)
{
    int i;
    for (;;) {
        i = producir(id);
        wait(e);
        wait(s);
        anyadir(i);
        signal(s);
        signal(n);
    }
}
void consumidor(char id)
{
    int i;
    for (;;) {
        wait(n);
        wait(s);
        i = tomar();
        signal(s);
        signal(e);
        consumir(id,i);
    }
}
main( )
{
    initialsem(s,1);
    initialsem(n,0);
    initialsem(e,TamanyoDeBuffer);
    initialsem(ts,1);
    cobegin {
        productor('A'); productor ('B');
        consumidor('x'); consumidor ('y'); consumidor ('z');
    }
}
// boundedbuff.inc – fichero a incluir para buffer acotado
const int TamanyoDeBuffer = 5;
int buffer[TamanyoDeBuffer];
int in = 0; // índice del buffer a usar para el siguiente añadir
int out = 0; // índice del buffer a usar para el siguiente tomar
```

```
void anyadir(int v)
    // añadir v al buffer
    // se asume que el desbordamiento se evita
    // mediante semáforos o condiciones externos
{
    buffer[in] = v;
    in = (in + 1) % TamanyoDeBuffer;
}
int tomar( )
    // devuelve un elemento del buffer
    // se asume que la extenuación se evita
    // mediante semáforos o condiciones externos
{
    int tmp;
    tmp = buffer[out];
    out = (out + 1) % TamanyoDeBuffer;
    return tmp;
}
```

El siguiente es un ejemplo de la ejecución de la solución precedente al *buffer* acotado en BACI.

```
Fichero fuente: semprodcons.cm Vie 1 Ago 12:36:55 1997
Productor B produce 4
Productor A produce 13
Productor B produce 12
Productor A produce 4
Productor B produce 17
Consumidor x consume 4
Consumidor y consume 13
Productor A produce 16
Productor B produce 11
Consumidor z consume 12
Consumidor x consume 4
Consumidor y consume 17
Productor B produce 6
. . .
```

E.4. PROYECTOS BACI

En esta sección, exploramos dos tipos generales de proyectos que pueden implementarse en BACI. Primero examinamos proyectos que involucran la implementación de operaciones de bajo nivel (ej., instrucciones máquina especiales que se utilizan para sincronizar el acceso de procesos a una

memoria principal común). Luego examinamos proyectos que se construyen sobre estas operaciones de bajo nivel (ej., problemas clásicos de sincronización). Puede encontrarse más información sobre estos proyectos en [BYNU96] y en las descripciones de proyectos incluidas en la distribución de BACI. Los profesores que deseen acceso a las soluciones de algunos de estos proyectos pueden contactar con los autores. Además de los proyectos expuestos en esta sección, muchos de los problemas propuestos al final del Capítulo 5 y del Apéndice A pueden ser implementados en BACI.

IMPLEMENTACIÓN DE PRIMITIVAS DE SINCRONIZACIÓN

Implementación de instrucciones máquina. Hay numerosas instrucciones máquina que pueden ser implementadas en BACI. Por ejemplo, puede implementarse la instrucción *exchange* (intercambio) dada en la Figura 5.2 o la operación *compare-and-swap* (comparar e intercambiar) descrita en [HERL90]. La implementación de estas instrucciones debe basarse en una función atómica que devuelva un valor entero. Puede ponerse a prueba la implementación de la instrucción máquina construyendo un protocolo de exclusión mutua en base a la operación de bajo nivel.

Implementación de semáforos imparciales (FIFO). La manipulación de semáforos en BACI está implementada como un orden de extracción aleatorio, que es como los semáforos fueron definidos originalmente por Dijkstra. Como se indicó en la Sección 5.3, la política más justa es la FIFO. Se pueden implementar semáforos con política de extracción FIFO en BACI. Como mínimo la implementación debería definir los siguientes cuatro procedimientos:

- `CreateSemaphores()` para inicializar el código del programa.
- `InitSemaphore(int sem_index)` para inicializar el semáforo representado por `sem_index`.
- `FIFOP(int sem_index)`
- `FIFOV(int sem_index)`

Este código precisa ser escrito como una implementación del sistema y, por tanto, debe controlar todos los posibles errores. En otras palabras, el diseñador del semáforo es responsable de producir código robusto que admita ser utilizado por usuarios ignorantes o estúpidos e incluso maliciosos.

SEMÁFOROS, MONITORES E IMPLEMENTACIONES

Hay muchos problemas clásicos de programación concurrente: el problema productor/consumidor, los filósofos comensales, el problema de los lectores/escritores con diferentes prioridades, el problema de la barbería y el problema de los fumadores. Todos estos problemas pueden ser implementados en BACI. En esta sección, se exponen proyectos no estándar de semáforos y monitores que pueden ser implementados en BACI para ayudar a un mejor entendimiento de los conceptos de concurrencia y sincronización.

Aes y Bes y Semáforos. Para el siguiente esbozo de programa en BACI,

```
// aquí las declaraciones de semáforos globales
void A( )
{
```

```
    SOLAMENTE llamadas a p( ) y v( )
}
void B( )
{
    SOLAMENTE llamadas a p( ) y v( )
}
main( )
{
    // aquí la inicialización de los semáforos
    cobegin {
        A( ); A( ); A( ); B( ); B( );
    }
}
```

complete el programa utilizando el mínimo número de semáforos generales, de manera que los procesos terminen SIEMPRE en el orden A (cualquier copia), B (cualquier copia), A (cualquier copia), A, B. Utilice la opción `-t` del intérprete para mostrar la terminación de los procesos. (Existen muchas variaciones a este proyecto. Por ejemplo, hacer que cuatro procesos concurrentes terminen en el orden ABAA o que ocho procesos concurrentes terminen en el orden AABABABB).

Utilizando semáforos binarios. Repita el proyecto anterior utilizando semáforos binarios. Explique por qué para esta solución se necesitan sentencias IF-THEN-ELSE y de asignación, si bien no fueron necesarias en la solución al proyecto previo. En otras palabras, explique por qué en este caso no se puede utilizar solamente P y V.

Espera activa versus semáforos. Compare las prestaciones de una solución a la exclusión mutua que utilice espera activa (ej. la instrucción testset) con una solución que utilice semáforos. Por ejemplo, compare una solución con semáforos y una solución con testset al proyecto ABAAB expuesto previamente. En cada caso, realice un gran número de ejecuciones (digamos 1000) para obtener mejores estadísticas. Razone los resultados obtenidos, explicando por qué una implementación es preferible a la otra.

Semáforos y monitores. Con el espíritu del Problema 5.17, implemente en BACI un monitor utilizando semáforos generales y luego implemente un semáforo general usando un monitor.

Semáforos generales y binarios. Pruebe que los semáforos generales y los semáforos binarios son igual de potentes, implementando un tipo de semáforo con el otro y viceversa.

Tictac de Reloj. Un Proyecto Monitor. De forma similar al Problema 7.17 en [SILB02], escriba un programa que contenga un monitor RelojDespertador. El monitor debe tener una variable `int` llamada `ElReloj` (inicializada a cero) y dos funciones:

- `Tic()`. Esta función incrementa `ElReloj` cada vez que se le llama. También puede hacer otras cosas, como `signalc`, si es necesario.
- `int Alarma(int id, int delta)`. Esta función bloquea al llamante con identificador `id` durante como mínimo `delta` tics de `ElReloj`.

El programa principal también debe tener dos funciones:

- `void TicTac()`. Este procedimiento llama a `Tic()` en un bucle infinito.
- `void Hilo(int id, int miDelta)`. Esta función llama a `Alarma()` en un bucle infinito.

Pueden dotar al monitor de cualesquiera otras variables que necesite. El monitor debe ser capaz de acomodar hasta cinco alarmas simultáneas.

Un problema sobre un panadero popular. Debido a la reciente popularidad de una panadería, casi todos los clientes tienen que esperar a ser servidos. Para gestionar el servicio, el panadero quiere instalar un sistema de tiquets que asegure que los clientes son servidos por turno. Construya una implementación BACI de este sistema de tiques.

E.5. MEJORAS AL SISTEMA BACI

El sistema BACI está siendo mejorado en diversos aspectos.

1. Se ha añadido una interfaz gráfica de usuario (GUI) a la versión UNIX de BACI. Esta interfaz permite al usuario editar, compilar e interpretar programas BACI todo en el mismo sistema. Ventanas coloreadas ilustran la ejecución de los programas BACI. Esta interfaz está disponible en http://www.mines.edu/fs_home/tcamp/GUI/. Si busca una GUI alternativa, siga leyendo.
2. Se ha creado una versión distribuida de BACI. De manera similar a como sucede con los programas concurrentes, es difícil probar la corrección de los programas distribuidos sin una implementación. El BACI distribuido permitirá la fácil implementación de programas distribuidos. Además de para probar la corrección de un programa distribuido, el BACI distribuido puede utilizarse para comprobar las prestaciones del programa. BACI distribuido está disponible en http://www.mines.edu/fs_home/tcamp/dbaci/.
3. Se dispone de un desensamblador de PCODE que le proporciona al usuario un listado anotado de un fichero PCODE, mostrando los mnemónicos de cada instrucción y, si está disponible, el correspondiente fragmento del programa fuente que generó la instrucción. Este desensamblador está incluido en el sistema BACI.
4. Se ha añadido la capacidad de compilación separada y de variables externas a ambos compiladores. El sistema BACI incluye un gestor de bibliotecas y un contador que permiten la creación y uso de bibliotecas de PCODE de BACI. Para más detalles véase la guía de usuario de compilación separada de BACI.
5. Actualmente está en construcción una implementación en Java del sistema BACI que se espera concluir en el verano del 2004. Esta versión del sistema BACI ejecutará sobre cualquier computador que tenga instalada una máquina virtual Java.

El sistema BACI también ha sido mejorado por otros.

1. David Strite, un estudiante que trabaja con Linda Null en la universidad del estado de Pennsylvania, ha creado un depurador de BACI, con una GUI. Este depurador está disponible en http://cs.hbg.psu.edu/~null/baci/.
2. Utilizando BACI y la GUI de BACI de la universidad del estado de Pennsylvania, Moti Ben-Ari del Instituto de Ciencia Weizmann en Israel, creó un entorno integrado de desarrollo para el aprendizaje de la programación concurrente mediante la concurrencia simulada denominado jBACI. Está disponible en http://stwww.weizmann.ac.il/g-cs/benari/jbaci/.

GLOSARIO

acceso directo a memoria (DMA) Una técnica de E/S en la que un módulo especial, llamado módulo de DMA, controla el intercambio de datos entre la memoria principal y un dispositivo de E/S. El procesador envía una solicitud de transferencia de un bloque de datos al módulo de DMA, siendo interrumpido sólo después de que se haya transferido el bloque completo.

acceso directo La capacidad de obtener datos o introducir datos en un dispositivo de almacenamiento en una secuencia independiente de su posición relativa, a través de direcciones que indican la ubicación física de los datos.

acceso indexado Perteneciente a la organización y acceso de los registros de una estructura de almacenamiento, a través de un índice separado de cada registro almacenado.

acceso secuencial indexado Perteneciente a la organización y acceso de los registros de una estructura de almacenamiento, a través de un índice de las claves que se almacenan en ficheros secuenciales arbitrariamente particionados.

acceso secuencial La capacidad de introducir datos en un dispositivo de almacenamiento o en un soporte de datos en la misma secuencia en la que estaban ordenados los datos, o de obtenerlos en el mismo orden en el que se introdujeron.

activar Reservar tiempo en el procesador para trabajos o tareas que se encuentran listas para su ejecución.

arquitectura de comunicaciones La estructura hardware y software que implementa las funciones de comunicaciones.

base de datos Una colección de datos interrelacionados, frecuentemente con control de redundancia y organizados en base a un esquema que sirve a una o más aplicaciones; los datos se almacenan de forma que puedan utilizarlos diferentes programas sin preocuparse de la estructura u organización de los datos. Se utiliza una técnica común para añadir nuevos datos y para modificar y obtener datos existentes.

Beowulf Define una clase de computación *cluster* que se centra en minimizar la tasa precio-prestaciones sin comprometer su rendimiento para realizar el trabajo de computación para el que fue construido. La mayor parte de los Sistemas Beowulf están implementados en computadores Linux.

bloque (1) Un conjunto de registros contiguos que se graban como una unidad; las unidades quedan separadas por huecos entre registros.

bloque (2) Un grupo de bits que se transmiten como una unidad.

bloque de control de proceso Manifestación de un proceso en el sistema operativo. Es una estructura de datos que contiene información sobre las características y estado de los procesos.

***buffer* de traducción anticipada (*translation lookaside buffer*, TLB)** Una memoria cache ed alta velocidad usada para almacenar las referencias a las entradas de las tablas de páginas que se han hecho recientemente como parte del esquema de memoria virtual. La TLB reduce la frecuencia de accesos a memoria principal para recuperar entradas de la tabla de páginas.

bus del sistema Un bus usado para interconectar los principales componentes del computador (procesador, memoria y E/S).

buzón Una estructura de datos compartida entre varios procesos que se utiliza como una cola de mensajes. Los mensajes se envían al buzón y se extraen del buzón en vez de pasar directamente del emisor al receptor.

***cache* de disco** Un *buffer*, habitualmente almacenado en memoria principal, que funciona como una cache de bloques de disco entre el almacenamiento en disco y el resto de la memoria principal.

cambio de modo Una operación hardware que cuando se activa hace que el procesador en un modo diferente (kernel o usuario). Cuando el modo pasa de usuario a kernel, se guardan el contador de programa, la palabra de estado del procesador, y otros registros. Cuando le modo cambia de kernel a usuario, se restaura esta información.

cambio de proceso Operación que cambia el procesador de un proceso a otro, guardado el bloque de control de proceso, los registros, y otra información del primero y reemplazándolo con la información del segundo proceso.

campo (1) Datos lógicos definidos que son parte de un registro.

campo (2) La unidad elemental de un registro que puede contener un elemento de datos, un conjunto de datos, un puntero o un enlace.

caparazón, interfaz de mandatos La porción del sistema operativo que interpreta mandatos de usuario interactivos y mandatos de lenguaje de control de trabajos. Funciona como una interfaz entre el usuario y el sistema operativo.

cerrojo cíclico Un mecanismo de exclusión mutua en el que un proceso ejecuta un bucle infinito esperando hasta que el valor de una variable que actúa de cerrojo indique que está disponible.

ciclo de instrucción El periodo de tiempo que comprende la lectura de memoria de la instrucción y su ejecución cuando un computador procesa una instrucción en lenguaje máquina.

círculo vicioso (*livelock*) Una condición en la cual dos o más procesos cambian continuamente su estado en respuesta a cambios en los otros procesos sin realizar ningún trabajo útil. Es similar al interbloqueo (*deadlock*) en que no se consigue ningún progreso pero difiere en que ningún proceso se bloquea o espera por nada.

cita En paso de mensajes, una condición en la cual emisor y receptor de un mensaje están ambos bloqueados hasta que el mensaje se entrega.

cliente Un proceso que solicita servicios a procesos servidores mediante el envío de mensajes.

cluster Grupo de *computadoras completas* e interconectadas, que trabajan juntas como un recurso de computación unificado y que pueden crear la ilusión de ser una única máquina. El término *computadora completa* significa un sistema que puede ejecutar por sí mismo, aparte del clúster.

compactación Una técnica utilizada cuando la memoria se divide en particiones de tamaño variable. De vez en cuando, el sistema operativo desplaza las particiones de forma que queden contiguas y la memoria libre quede en un único bloque libre. Véase *fragmentación externa*.

compartición de tiempo El uso concurrente de un dispositivo por parte de varios usuarios.

concurrente Relativo a procesos o hilos que tienen lugar dentro de un intervalo común de tiempo durante el cual pueden tener que compartir alternativamente recursos comunes.

condición de carrera Situación en la cual múltiples procesos acceden y manipulan datos compartidos de manera que el resultado depende de la velocidad relativa de los procesos.

conjunto de trabajo El conjunto de trabajo con parámetro D de un proceso en un tiempo t, W(t, D), es el conjunto de páginas a las que el proceso hace referencia en las últimas D unidades de tiempo. Compárese con el *conjunto residente*.

conjunto residente La parte del proceso que está realmente en memoria en un instante determinado. Compárese con el *conjunto residente*.

contador de programa Registro de dirección de la instrucción.

contexto de ejecución Lo mismo que *estado de proceso*.

cortar el tiempo Un modo de operación en el que se asignan rodajas de tiempo a dos o más procesos en el mismo computador.

creación de procesos Creación de un nuevo proceso por parte de otro.

criptografía La conversión de texto plano o datos a algo ininteligible por medio de un cómputo matemático reversible.

descriptor de proceso Idéntico al bloque de control de proceso.

detección del interbloqueo Una técnica en la que los recursos solicitados siempre se asignan si están disponibles. Periódicamente, el sistema operativo comprueba la existencia del interbloqueo.

dirección base Una dirección que se usa como el origen en el cálculo de las direcciones durante la ejecución de un programa del computador.

dirección física La ubicación absoluta de una unidad de datos en memoria (por ejemplo, la palabra o byte en memoria principal, el bloque en memoria secundaria).

dirección lógica Una referencia a una ubicación de memoria independiente de la asignación actual de los datos a memoria. Antes de acceder a memoria, se debe llevar a cabo una traducción a una dirección física.

dirección real Una dirección física de memoria principal.

dirección relativa Una dirección calculada como un desplazamiento respecto a una dirección base.

dirección virtual La dirección de una ubicación en memoria virtual.

E/S programada Una técnica de E/S en la que el procesador envía un mandato de E/S a un módulo de E/S y debe, a continuación, esperar hasta que se complete la operación antes de continuar.

espacio de direcciones El rango de direcciones disponible para un programa de computador.

espera activa La ejecución repetida de un bucle de código mientras se espera a que ocurra un evento.

estado del proceso Toda la información que el sistema operativo necesita para gestionar un proceso y que el procesador necesita para ejecutar adecuadamente un proceso. El estado del proceso incluye los contenidos de varios registros del procesador, tales como el contador de programa y los registros de datos; también incluye información de uso del sistema operativo, tales como la prioridad del proceso y si el proceso está esperando por la finalización de un determinado evento de E/S. Lo mismo que *contexto de ejecución*.

estado no privilegiado Un contexto de ejecución que no permite que se ejecuten instrucciones hardware sensibles, como, por ejemplo, una instrucción de parada o instrucciones de E/S.

exclusión mutua Una condición en la cual hay un conjunto de procesos, de los cuales sólo uno a la vez es capaz de acceder a un recurso dado para realizar una función dada. Véase *sección crítica*.

expropiación Quitar un recurso a un proceso antes de que éste haya terminado de usarlo.

fallo de página Ocurre cuando una página que contiene una palabra a la que se está referenciando no se encuentra en memoria principal. Esto dispara una excepción y se requiere que se traiga a memoria dicha página.

fichero de acceso directo o *hash* Un fichero que contiene registros que se pueden acceder de acuerdo a los valores de un campo clave. Se utiliza un método *hash* para localizar un registro en base a su valor clave.

fichero indexado Un fichero cuyos registros se acceden de acuerdo al valor de campos clave. Se requiere un índice que indique la ubicación de cada registro en función de cada valor clave.

fichero secuencial indexado Un fichero cuyos registros se ordenan de acuerdo a los valores de un campo clave. El fichero principal tiene un fichero índice que contiene una lista parcial de valores clave; el índice proporciona una capacidad de búsqueda que permite encontrar fácilmente la vecindad de un determinado registro.

fichero secuencial Un fichero cuyos registros se ordenan de acuerdo a los valores de uno o más campos clave y se procesan en la misma secuencia desde el principio del fichero.

fichero Un conjunto de registros tratados como una unidad.

fragmentación externa Ocurre cuando la memoria se divide en particiones de tamaño variable correspondientes a los bloques de datos asignados a la memoria (por ejemplo, segmentos en memoria principal). Debido a que los segmentos se mueven en memoria, se crean huecos entre las porciones de memoria ocupadas.

fragmentación interna Ocurre cuando la memoria se divide en particiones de tamaño fijo (por ejemplo, marcos de páginas en memoria principal, bloques físicos en disco). Si se asigna un bloque de datos a una o más particiones, se podría malgastar espacio en la última partición. Esto ocurrirá si la última porción de datos es más pequeña que la última partición.

grado de multiprogramación El número de procesos que están parcial o totalmente en memoria principal.

gusano Programa que puede viajar de ordenador a ordenador a través de conexiones de red. Puede contener un virus o una bacteria.

hambruna Una condición en la cual un proceso se postpone indefinidamente porque otros procesos tienen siempre preferencia.

hashing Selección de la posición de almacenamiento de un objeto de datos por medio del cálculo de la dirección como una función del contenido de los datos. Esta técnica complica la función de reserva de almacenamiento pero proporciona una recuperación directa de los datos muy rápida.

hilo Una unidad de trabajo que se puede planificar. Incluye un contexto de procesador (que incluye el contador de programa y el puntero de pila) y su propia área de pila (para permitir el salto entre subrutinas). Un hilo ejecuta secuencialmente y el procesador puede interrumpirlo para ceder el control a otro hilo. Un proceso puede estar formado por múltiples hilos.

imagen de un proceso Todos los ingredientes de un proceso, incluyendo el programa, datos, pila, y el bloque de control de proceso.

instrucción privilegiada Una instrucción que sólo se puede ejecutar en un determinado modo; habitualmente la ejecuta un programa de supervisión.

interbloqueo (1) Una situación de bloqueo indefinido que se produce cuando hay múltiples procesos que están esperando que quede disponible un recurso que nunca estará disponible ya que lo tiene asignado otro proceso que está en un estado de espera similar.

interbloqueo (2) Una situación de bloqueo indefinido que se produce cuando hay múltiples procesos que están esperando que se produzca una acción o una respuesta de otro proceso que está en un estado de espera similar.

intercambio de hilos El acto de cambiar el control del procesador de un hilo a otro del mismo proceso.

interfaz de programación de aplicaciones (*application programming interface*, API) Una biblioteca estándar de herramientas de programación utilizada por los desarrolladores de software para escribir aplicaciones compatibles con un sistema operativo o interfaz gráfica específicos.

interrupción habilitada Una condición, usualmente causada por el sistema operativo, durante la cual el procesador responderá a las señales de petición de interrupción de un determinado tipo.

interrupción inhabilitada Una condición, usualmente causada por el sistema operativo, durante la cual el procesador ignorará las señales de petición de interrupción de un determinado tipo.

interrupción Una suspensión de un proceso, como puede ser la ejecución de un programa del computador, causada por un evento externo a ese proceso y realizada de manera que la ejecución del proceso pueda ser reanudada.

inversión de prioridad Una circunstancia en la cual el sistema operativo fuerza a una tarea de mayor prioridad a esperar por una tarea de menor prioridad.

lenguaje de control de trabajos (*job control language,* JCL) Un lenguaje orientado a problemas que está diseñado para expresar sentencias utilizadas para identificar el trabajo o describir sus requisitos para el sistema operativo.

lista encadenada Una lista en la que los elementos pueden estar dispersos, pero en la que cada elemento contiene un identificador para localizar al siguiente.

llamada a procedimiento remoto (*remote procedure call,* RPC) Una técnica por la cual dos programas en diferentes máquinas interaccionan utilizando sintaxis y semántica *call/return*. Ambos programas (el que invoca el procedimiento y el que es invocado) se comportan como si se ejecutaran en la misma máquina.

macronúcleo Parte central, de gran tamaño, de un sistema operativo que proporciona una gran variedad de servicios.

manejador de dispositivo Un módulo del sistema operativo (normalmente del núcleo) que trata directamente con un dispositivo o módulo de E/S.

manejador de interrupción Una rutina que, generalmente, forma parte del sistema operativo. Cuando se produce una interrupción, se transfiere el control al manejador de interrupción, que realiza alguna acción en respuesta a la condición que causó la interrupción.

marco de página Un bloque contiguo de memoria principal de tamaño fijo que se utiliza para contener una página.

marco En almacenamiento virtual paginado, un bloque de longitud fija de memoria principal que se utiliza para contener una página de memoria virtual.

mediador de solicitud de objeto (*object request broker,* ORB) Una entidad en un sistema orientado a objetos que actúa como intermediario de las solicitudes enviadas de un cliente a un servidor.

memoria cache Una memoria más pequeña y rápida que la memoria principal, que se interpone entre el procesador y la memoria principal. La cache actúa como un *buffer* de las posiciones de memoria recientemente usadas.

memoria principal La memoria interna del computador, que es accesible a los programas especificando direcciones, y que puede cargarse en registros para la posterior ejecución o procesamiento.

memoria secundaria Una memoria situada fuera del propio computador; es decir, el procesador no puede accederla directamente. Primero, debe copiarse a la memoria principal. El disco y la cinta son ejemplos de este tipo de memoria.

memoria virtual El espacio de almacenamiento direccionable, en el cual las direcciones virtuales se traducen a direcciones reales. El tamaño del almacenamiento virtual está limitado por el esquema de direccionamiento del sistema de computación y por la cantidad de memoria secundaria disponible y no por el tamaño de memoria principal.

mensaje Un bloque de información que puede ser intercambiada entre procesos por medio de comunicación.

método de acceso Un método utilizado para encontrar un fichero, un registro o un conjunto de registros.

micronúcleo Parte central, de pequeño tamaño, de un sistema operativo que proporciona planificación de procesos, gestión de memoria y servicios de comunicación y depende de otros procesos para realizar alguna de las funciones tradicionalmente asociadas con el núcleo del sistema operativo.

migración de proceso La transferencia de suficiente cantidad del estado de un proceso de una máquina a otra para que el proceso ejecute en la máquina destino.

modo núcleo Un modo privilegiado de ejecución reservado para el núcleo del sistema operativo. Normalmente, en modo núcleo se puede acceder a regiones de memoria no disponibles para los procesos que ejecutan en un modo menos privilegiado. También permite la ejecución de ciertas instrucciones de máquina restringidas al modo núcleo. También se denomina *modo sistema* o *modo privilegiado*.

modo privilegiado Igual que *modo núcleo*.

modo sistema Lo mismo que *modo núcleo*.

modo usuario El modo de ejecución menos privilegiado. No se pueden utilizar ciertas regiones de memoria y ciertas instrucciones de máquina en este modo.

monitor Una construcción del lenguaje de programación que encapsula variables, procedimientos de acceso y código de inicialización dentro de un tipo abstracto de datos. La variable del monitor sólo puede ser accedida vía sus procedimientos de acceso y sólo un proceso a la vez puede estar accediendo activamente al monitor. Los procedimientos de acceso son secciones críticas. Un monitor puede tener una cola de procesos esperando para acceder.

multiprocesador con acceso a memoria no uniforme (*nonuniform memory access*, NUMA) Un procesador de memoria compartida en el cual el tiempo de acceso de un procesador a una palabra de memoria determinada varía dependiendo de la localización en memoria de la misma.

multiprocesador Un computador que tiene dos o más procesadores con acceso común al almacenamiento principal.

multiprocesamiento simétrico (SMP) Una forma de multiprocesamiento que permite a las aplicaciones ejecutar en cualquier procesador disponible o en varios procesadores disponibles al mismo tiempo.

multiproceso Un modo de operación que permite el procesamiento paralelo mediante dos o más procesadores de un multiprocesador.

multiprogramación Un modo de operación que permite la ejecución intercalada de dos o más programas en un único procesador. Lo mismo que multitarea, utilizando diferente terminología.

multitarea Un modo de operación que permite la ejecución intercalada de dos o más tareas de computación. Lo mismo que multiprogramación, utilizando diferente terminología.

mutex Un semáforo binario.

núcleo monolítico Un gran núcleo que contiene prácticamente el sistema operativo completo, incluyendo la planificación, sistema de ficheros, manejadores de dispositivos y gestión de memoria. Todos los componentes funcionales del núcleo tienen acceso a todas las estructuras de datos y rutinas internas. Normalmente, un núcleo monolítico se implementa como un único proceso, con todos los elementos compartiendo el mismo espacio de direcciones.

núcleo Una porción del sistema operativo que incluye las porciones de software utilizadas más frecuentemente. Generalmente, el núcleo se mantiene permanentemente en memoria principal. El núcleo ejecuta en modo privilegiado y responde a llamadas de procesos e interrupciones de dispositivos.

operación asíncrona Una operación que ocurre sin una relación temporal regular o predecible con un determinado evento; por ejemplo, la invocación de una rutina de diagnóstico que puede recibir el control en cualquier momento durante la ejecución de un programa del computador.

operación síncrona Una operación que ocurre de forma regular o predecible con respecto a la aparición de un determinado evento en otro proceso; por ejemplo, la invocación de una rutina de entrada/salida que recibe el control en una posición prefijada en un programa del computador

organización de ficheros El orden físico de los registros en un fichero, que determina el método de acceso utilizado para almacenarlos y obtenerlos.

página En almacenamiento virtual, un bloque de longitud fija que tiene una dirección virtual y que se transfiere como una unidad entre memoria principal y secundaria.

paginación adelantada Recuperación de páginas adicionales a la que ha causado un fallo de página. La esperanza es que las páginas extra se vayan a usar en un futuro próximo, aprovechando así la E/S de disco. Compárese con *paginación bajo demanda*.

paginación bajo demanda La transferencia de una página de memoria secundaria memoria principal se hace únicamente en el momento en el que se necesite. Compárese con *paginación adelantada*.

paginación La transferencia de páginas entre memoria principal y secundaria.

palabra de estado del programa (PSW) Un registro o conjunto de registros que contiene códigos de condición, modo de ejecución y otra información de estado que refleja el estado del proceso.

palabra Un conjunto ordenado de bytes o bits que constituyen la unidad normal en la que se almacena, transmite u opera la información en un determinado computador. Normalmente, si un computador tiene un juego de instrucciones de tamaño fijo, la longitud de la instrucción es igual al de la palabra.

particionamiento de memoria La división de almacenamiento en secciones independientes.

pila Una lista ordenada en la que los elementos se añaden y eliminan del mismo extremo de la lista, conocido como cima. Es decir, el próximo elemento que se añade a la lista se coloca en la cima, mientras que el siguiente elemento que se eliminará de la lista es el que ha estado en ella durante menos tiempo. Este método se caracteriza como el último en entrar-el primero en salir.

planificación en pandilla La planificación de un conjunto de hilos relacionados para que ejecuten a la vez sobre un conjunto de procesadores, en una relación uno a uno.

planificación Seleccionar trabajos o tareas que van a ser activados. En algunos sistemas operativos, otras unidades de trabajo, tales como operaciones de entrada/salida, también podrían ser planificadas.

predicción del interbloqueo Una técnica dinámica que analiza cada solicitud de un nuevo recurso, con respecto a la posibilidad de interbloqueo. Si la nueva solicitud puede conducir a un interbloqueo, la solicitud se deniega.

prevención del interbloqueo Una técnica que garantiza que nunca se producirá un interbloqueo. La prevención se logra asegurando que no se cumple una de las condiciones necesarias.

primero entra primero sale Una técnica de colas en que el siguiente elemento en ser atendido es el elemento que ha estado en la cola mayor tiempo.

primero llega primero se atiende Lo mismo que *FIFO*.

procedimiento reentrante Una rutina que puede ser invocada antes de que se complete una ejecución previa de la misma, ejecutándose correctamente.

procesador En un computador, una unidad funcional que interpreta y ejecuta instrucciones. Un procesador consta de al menos una unidad de control de instrucciones y una unidad aritmética.

procesamiento en lotes o *batch* Perteneciente a la técnica de ejecución de un conjunto de programas, en el cual cada programa se ejecuta justo a continuación de que termine el programa anterior.

proceso ligero Un hilo.

proceso Un programa en ejecución. El sistema operativo se encarga de controlar y planificar un proceso. Lo mismo que *tarea*.

proximidad de referencias La tendencia de un procesador a acceder al mismo conjunto de posiciones de memoria repetidamente durante un corto periodo de tiempo.

puerta secreta Punto de entrada secreto y no documentado a un programa, usado para conseguir el acceso sin los métodos habituales de autenticación.

recurso consumible Un recurso que se puede crear (producir) y destruir (consumir). Cuando un proceso adquiere un recurso de este tipo, el recurso deja de existir. Las interrupciones, las señales, los mensajes y la información en *buffers* de E/S constituyen ejemplos de recursos consumibles.

recurso reutilizable Un recurso que sólo puede usarlo de forma segura un único proceso en cada momento y no se destruye cuando se usa. Los procesos obtienen unidades de recursos reutilizables que, posteriormente, liberan para que los usen otros procesos. Algunos ejemplos de este tipo de recursos son los procesadores, los canales de E/S, la memoria primaria y secundaria, los dispositivos y estructuras de datos como ficheros, bases de datos y semáforos.

registro lógico Un registro independiente de su entorno físico; se pueden localizar porciones de un registro lógico en diferentes registros físicos y varios registros lógicos o parte de régistros lógicos en un registro físico.

registro Un grupo de elementos de datos tratados como una unidad.

registros Una memoria de alta velocidad interna al procesador. Algunos registros son visibles para el usuario; es decir, disponibles para el programador mediante el juego de instrucciones de la máquina. Otros registros los usa sólo el procesador, para propósitos de control.

reubicación dinámica Un proceso que asigna direcciones absolutas nuevas a un programa durante su ejecución, de tal forma que pueda ejecutarse en un área de memoria principal diferente.

rodaja de tiempo El máximo tiempo que un proceso se ejecuta sin ser interrumpido.

sección crítica En un procedimiento asíncrono de un programa, una parte que no puede ser ejecutada simultáneamente con una sección crítica asociada de otro procedimiento asíncrono. Véase *exclusión mutua*.

segmentación La división de un programa o aplicación en segmentos como parte de un esquema de memoria virtual.

segmento En memoria virtual, un bloque que tiene una dirección virtual. Los bloques de un programa pueden ser de diferente tamaño e incluso pueden variar su longitud de forma dinámica.

seguridad multinivel Una funcionalidad que obliga a que se hagan controles de acceso a través de múltiples niveles de clasificación de los datos.

semáforo binario Un semáforo que toma sólo los valores 0 y 1. Un semáforo binario permite que sólo un proceso o hilo a la vez tenga acceso al recurso crítico compartido.

semáforo débil Un semáforo en el cual todos los procesos que esperan por él avanzarán en un orden no especificado (ej., el orden no se conoce o es indeterminado).

semáforo fuerte Un semáforo en el cual todos los procesos que esperan por él, se encolan, y serán eventualmente ejecutados, en el mismo orden en que ejecutaron las operaciones wait (P), (orden FIFO).

semáforo Un valor entero utilizado para la señalización entre procesos. Sobre un semáforo sólo pueden realizarse tres operaciones, todas ellas atómicas: inicialización, decremento e incremento. Dependiendo de la definición exacta del semáforo, la operación de decremento puede provocar el bloqueo de un proceso, y la operación de incremento puede provocar el desbloqueo de un proceso. También conocido como **semáforo con contador** o **semáforo general**.

servidor (1) Un proceso que responde a solicitudes de clientes vía mensajes.

servidor (2) En una red, una estación que proporciona servicios a otras estaciones; por ejemplo, un servidor de ficheros, un servidor de impresión o un servidor de correo.

sesión Una colección de uno o más procesos que representan una única aplicación interactiva de usuario o una función del sistema operativo. Todas las entradas del teclado y del ratón se dirigen a la sesión en primer plano y todas las salidas de las sesiones en segundo plano se dirigen a la pantalla.

sincronización Situación en que dos o más procesos coordinan sus actividades basándose en una condición.

sistema confiable Un ordenador y un sistema operativo que puede verificarse si implementa una política de seguridad dada.

sistema de gestión de ficheros Un conjunto de programas de sistema que proporcionan servicios a los usuarios y las aplicaciones para el uso de los ficheros, incluyendo acceso a los ficheros, mantenimiento de directorios y control de acceso.

sistema de tiempo real Un sistema operativo que debe planificar y gestionar tareas de tiempo real.

sistema operativo distribuido Un sistema operativo común compartido por una red de computadores. El sistema operativo distribuido da soporte a la comunicación entre procesos, migración de procesos, exclusión mutua y la prevención o detección de interbloqueos.

sistema operativo El módulo software que controla la ejecución de los programas y que proporciona servicios tales como asignación de recursos, planificación, control de E/S y gestión de datos.

software malicioso Cualquier software diseñado para causar daño o para usar recursos de sistema del equipo afectado. El software malicioso (*malware*) frecuentemente se oculta dentro o se hace pasar por software legítimo. En algunos caso se expande él mismo a otros ordenadores por medio de correo electrónico o disquetes infectados. Los diferentes tipos de software malicioso son los virus, troyanos, gusanos o software oculto que realiza ataques de denegación de servicio.

spooling El uso de memoria secundaria como un *buffer* de almacenamiento para reducir los retardos de procesamiento cuando se transfieren datos entre equipos periféricos y los procesadores de un computador.

swapping Un proceso que intercambia los contenidos de un area de memoria principal con un area de memoria secundaria.

tabla de asignación de disco Una tabla que indica qué bloques de almacenamiento secundario están libres y disponibles para su asignación a ficheros.

tabla de asignación de ficheros (*file allocation table*, FAT) Una tabla que indica la ubicación física en almacenamiento secundario del espacio asignado al fichero. Hay una tabla de asignación de ficheros por cada fichero.

tarea Lo mismo que *proceso*.

tareas de tiempo real Una tarea que se ejecuta en conexión con algún proceso o función o conjunto de eventos externos al sistema de computo y que debe cumplir uno o más plazos para interactuar efectiva y correctamente con el entorno externo.

tasa de aciertos En una memoria de dos niveles, la fracción de todos los accesos a memoria que se encuentran en la memoria más rápida (p. ej. la cache)

tiempo de ciclo de memoria El tiempo que tarda en leerse o escribirse una palabra de memoria. Es el inverso de la velocidad a la que se leen o escriben las palabras de memoria.

tiempo de respuesta Para un proceso interactivo, es el tiempo que trascurre desde que se lanza una petición hasta que se comienza a recibir la respuesta.

trabajo Un conjunto de pasos computacionales empaquetados para ejecutarse como una unidad.

traductor de direcciones Una unidad funcional que transforma direcciones virtuales en direcciones reales.

trap Salto condicional no programado a una dirección específica que el hardware activa automáticamente; se almacena la posición desde donde se ha hecho el salto.

trasiego o *thrasing* Fenómeno que se da en los esquemas de memoria virtual, en el cual el proceso consume más tiempo haciendo *swapping* de diferentes porciones de memoria que ejecutando instrucciones.

traza Secuencia de instrucciones que se ejecutan cuando un proceso está corriendo.

troyano Rutina secreta no documentada insertada dentro de un programa de utilidad. La ejecución del programa hace que dicha rutina también se ejecute.

tubería Un *buffer* circular que permite que dos procesos se comuniquen según el modelo productor-consumidor. Por tanto, es una cola con una política de primero en entrar-primero en salir, escrita por un proceso y leída por otro. En algunos sistemas, la tubería se generaliza para permitir que se seleccione cualquier elemento de la cola para su consumo.

turno rotatorio Un algoritmo de planificación en que los procesos se activan de forma cíclica; es decir, los procesos están en una cola circular. Un proceso que no puede continuar ejecutando porque está esperando por algún evento (por ejemplo, terminación de un proceso hijo o una operación de entrada/salida) devuelve el control al planificador.

último en entrar primero en salir (LIFO) Una técnica de gestión de una cola en la que el próximo elemento que se recupera es el más recientemente incluido en la cola.

unidad central de proceso (CPU) La parte de un computador que lee y ejecuta instrucciones. Consta de una unidad aritmético-lógica (*Arithmetic and Logic Unit*, ALU), una unidad de control y registros. Habitualmente, se le denomina simplemente procesador.

virus Rutina secreta no documentada insertada dentro de un programa de utilidad. La ejecución del programa hace que dicha rutina también se ejecute.

REFERENCIAS

ABREVIATURAS

ACM *Association for Computing Machinery*

IEEE *Institute of Electrical and Electronics Engineers* (Instituto de Ingenieros Eléctricos y Electrónicos)

IRE *Institute of Radio Engineers* (Instituto de Ingenieros de Radio)

ABRA87 Abrams, M., y Podell, H. *Computer and Network Security*. Los Alamitos, CA: IEEE Computer Society Press, 1987.

ADAM92 Adam, J. «Virus Threats and Countermeasures.» *IEEE Spectru*m, Agosto 1992.

AGAR89 Agarwal, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Boston: Kluwer Academic Publishers, 1989.

ALVA90 Alvare, A. «How Crackers Crack Passwords or What Passwords to Avoid.» *Proceedings, UNIX Security Workshop I*I, Agosto 1990.

ANAN92 Ananda, A.; Tay, B.; y Koh, E. «A Survey of Asynchronous Remote Procedure Calls.» *Operating Systems Review*, Abril 1992.

ANDE80 Anderson, J. *Computer Security Threat Monitoring and Surveillance*. Fort Washington, PA: James P. Anderson Co., Abril 1980.

ANDE89 Anderson, T.; Laxowska, E.; y Levy, H. «The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.» *IEEE Transactions on Computer*s, Diciembre 1989.

ANDE97 Anderson,T.; Bershad, B.; Lazowska, E.; y Levy, H. «Thread Management for Shared-Memory Multiprocessors.» en [TUCK97]

ANDR90 Andrianoff, S. «A Module on Distributed Systems for the Operating System Course.» *Proceedings, Twenty-First SIGCSE Technical Symposium on Computer Science Education, SIGSCE Bulleti*n, Febrero 1990.

ARDE80 Arden, B., editor. *What Can Be Automated?* Cambridge, MA: MIT Press, 1980.

ARTS89a Artsy, Y., ed. Special Issue on Process Migration. *Newsletter of the IEEE Computer Society Technical Committee on Operating System*s, Invierno 1989.

ARTS89b Artsy, Y. «Designing a Process Migration Facility: The Charlotte Experience.» *Compute*r, Septiembre 1989.

ATLA89 Atlas, A., y Blundon, B. «Time to Reach for It All.» *UNIX Revie*w, Enero 1989.

AXFO88 Axford, T. *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*. New York: Wiley, 1988.

BACH86 Bach, M. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice Hall, 1986.

BACO03 Bacon, J., y Harris, T. *Operating Systems: Concurrent and Distributed SoftwareDesign*. Reading, MA:Addison-Wesley, 1998.

BAEN97 Baentsch, M., et al. «Enhancing the Web's Infrastructure: From Caching to Replication.» *Internet Computing*, Marzo/Abril 1997.

BAER80 Baer, J. *Computer Systems Architecture*. Rockville, MD: Computer Science Press, 1980.

BAR00 Bar, M. *Linux Internals*. New York, McGraw-Hill, 2000.

BARB90 Barbosa,V. «Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs.» *IEEE Transactions on Software Engineering*, Noviembre 1990.

BARK89 Barkley, R., y Lee, T. «A Lazy Buddy System Bounded by Two Coalescing Delays per Class.» *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, Diciembre 1989.

BAYS77 Bays, C. «A Comparison of Next-Fit, First-Fit, and Best-Fit.» *Communications of the ACM*, Marzo 1977.

BECK90 Beck, L. *System Software*. Reading, MA:Addison-Wesley, 1990.

BELA66 Belady, L. «A Study of Replacement Algorithms for a Virtual Storage Computer.» *IBM Systems Journal*, No. 2, 1966.

BEN82 Ben-Ari, M. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall, 1982.

BEN90 Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

BEN98 Ben-Ari, M., y Burns, A. «Extreme Interleavings» *IEEE Concurrency*, Julio-Septiembre, 1998.

BIRR89 Birrell, A. *An Introduction to Programming with Threads*. SRC Research Report 35, Compaq Systems Research Center, Palo Alto, CA, Enero 1989. Dsiponible en http://www.research.compaq.com/SRC.

BLAC90 Black, D. «Scheduling Support for Concurrency and Parallelism in the Mach Operating System.» *Computer*, Mayo 1990.

BOEB85 Boebert, W.; Kain, R.; y Young, W. «Secure Computing: the Secure Ada Target Approach.» *Scientific Honeyweller*, Julio 1985. Reimpreso en [ABRA87].

BOLO89 Bolosky, W.; Fitzgerald, R.; y Scott, M.«Simple But Effective Techniques for NUMA Memory Management.» *Proceedings, Twelfth ACM Symposium on Operating Systems Principles*, Diciembre 1989.

BONW94 Bonwick, J. «An Object-Caching Memory Allocator.» *Proceedings, USENIX Summer Technical Conference*, 1994.

BORG90 Borg, A.; Kessler, R.; y Wall, D. «Generation and Analysis of Very Long Address Traces.» *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Mayo 1990.

BOSW03 Boswell, W. *Inside Windows Server 2003*. Reading, MA:Addison-Wesley, 2003.

BOVE03 Bovet, D., y Cesati, M. *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly, 2003.

BREN89 Brent, R. «Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation.» *ACM Transactions on Programming Languages and Systems*, Julio 1989.

BREW97 Brewer, E. «Clustering: Multiply and Conquer.» *Data Communications*, Julio 1997.

BRIA99 Briand, L, y Roy, D. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. Los Alamitos, CA: IEEE Computer Society Press, 1999.

BRIN73 Brinch Hansen, P. *Operating System Principles*. Englewood Cliffs, NJ: Prentice Hall, 1973.

BRIN01 Brinch Hansen, P. *Classic Operating Systems: From Batch Processing to Distributed Systems.* New York: Springer-Verlag, 2001.

BROW84 Brown, R.; Denning, P.; y Tichy, W. «Advanced Operating Systems.» *Compute*r, Octubre 1984.

BUHR95 Buhr, P., y Fortier, M. «Monitor Classification.» *ACM Computing Survey*s, Marzo 1995.

BUTT99 Buttazzo, G. «Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments. *IEEE Transactions on Computer*s, Octubre 1999.

BUYY99a Buyya, R. *High Performance Cluster Computing:Architectures and Systems.* Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High Performance Cluster Computing: Programming and Applications.* Upper Saddle River, NJ: Prentice Hall, 1999.

BYNU96 Bynum, B., y Camp, T. «After You, Alfonse: A Mutual Exclusion Toolkit,» *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Educatio*n, Febrero 1996.

CABR86 Cabrear, L. «The Influence of Workload on Load Balancing Strategies.» *USENIX Conference Proceeding*s, Verano 1986.

CAO96 Cao, P.; Felten, E.; Karlin, A.; y Li, K. «Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling.» *ACM Transactions on Computer System*s, Noviembre 1996.

CARR81 Carr, R., y Hennessey, J. «WSClock—A Simple and Efficient Algorithm for Virtual Memory Management.» *Proceedings of the Eighth Symposium on Operating System Principle*s.» Diciembre 1981.

CARR84 Carr, R. *Virtual Memory Management.* Ann Arbor, MI: UMI Research Press, 1984.

CARR89 Carriero, N., y Gelernter, D. «How to Write Parallel Programs: A Guide for the Perplexed.» *ACM Computing Survey*s, Septiembre 1989.

CARR01 Carr, S; Mayoo, J.; and Shene, C. «Race Conditions: A Case Study.» *The Journal of Computing in Small College*s, Octubre 2001.

CASA94 Casavant, T., y Singhal, M. *Distributed Computing Systems.* Los Alamitos, CA: IEEE Computer Society Press, 1994.

CASS01 Cass, S. «Anatomy of Malice.» *IEEE Spectru*m, Noviembre 2001.

CAST92 Castillo, C.; Flanagan, E.; y Wilkinson, N. «Object-Oriented Design and Programming.» *AT&T Technical Journa*l, Noviembre/Diciembre 1992.

CHAN85 Chandy, K., y Lamport, L. «Distributed Snapshots: Determining Global States of Distributed Systems.» *ACM Transactions on Computer System*s, Febrero 1985.

CHEN92 Chen, J.; Borg, A.; y Jouppi, N. «A Simulation Based Study of TLB Performance.» *Proceedings of the 19th Annual International Symposium on Computer Architectur*e, Mayo 1992.

CHEN94 Chen, P.; Lee, E.; Gibson, G.; Katz, R.; y Patterson, D. «RAID: High-Performance, Reliable Secondary Storage.» *ACM Computing Survey*s, Junio 1994.

CHEN96 Chen, S., y Towsley, D. «A Performance Evaluation of RAID Architectures.» *IEEE Transactions on Computer*s, Octubre 1996.

CHES97 Chess, D. «The Future of Viruses on the Internet.» *Proceedings, Virus Bulletin International Conferenc*e, Octubre 1997.

CHRI93 Christopher, W.; Procter, S.; y Anderson,T. «The Nachos Instructional Operating System.» *Proceedings, 1993 USENIX Winter Technical Conferenc*e, 1993.

CHU72 Chu, W., y Opderbeck, H. «The Page Fault Frequency Replacement Algorithm.» *Proceedings, Fall Joint Computer Conferenc*e, 1972.

CLAR85 Clark, D., y Emer, J. «Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement.» *ACM Transactions on Computer System*s, Febrero 1985.

CLAR98 Clarke, D., y Merusi, D. *System Software Programming:The Way Things Work.* Upper Saddle River, NJ: Prentice Hall, 1998.

CLER04 Clercq, J. *Windows Server 2003 Security Infrastructure: Core Security Features.* Burlington, MA: Digital Press, 2004.

COFF71 Coffman, E.; Elphick, M.; y Shoshani, A. «System Deadlocks.» *Computing Survey*s, Junio 1971.

COME84 Comer, D., y Fossum, T. *Operating System Design: The Xinu Approach.* Englewood Cliffs, NJ: Prentice Hall, 1984.

CONW63 Conway, M. «Design of a Separable Transition-Diagram Compiler.» *Communications of the AC*M, Julio 1963.

CONW67 Conway, R.; Maxwell, W.; y Miller, L. *Theory of Scheduling.* Reading, MA:Addison-Wesley, 1967. Reimpreso por Dover Publications, 2003.

COOP89 Cooper, J. *Computer and Communications Security: Strategies for the 1990*s. New York: McGraw-Hill, 1990.

CORB62 Corbato, F.; Merwin-Daggett, M.; y Dealey, R. «An Experimental Time-Sharing System.» *Proceedings of the 1962 Spring Joint Computer Conferenc*e, 1962. Reimpreso en [BRIN01].

CORB68 Corbato, F. «A Paging Experiment with the Multics System.» *MIT Project MAC Report MAC-M-38*4, Mayo 1968.

CORB96 Corbett, J. «Evaluating Deadlock Detection Methods for Concurrent Software.» *IEEE Transactions on Software Engineerin*g, Marzo 1996.

COX89 Cox, A., y Fowler, R. «The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM.» *Proceedings, Twelfth ACM Symposium on Operating Systems Principle*s, Diciembre 1989.

COX00 Cox, P., y Sheldon, T. *Windows NT Security Handbook.* New York: Osborne McGraw-Hill, 2000.

CUST93 Custer, H. *Inside the Windows NT.* Redmond, WA: Microsoft Press, 1993.

CUST94 Custer, H. *Inside the Windows NT File System.* Redmond, WA: Microsoft Press, 1994.

DALE68 Daley, R, y Dennis, R. «Virtual Memory, Processes, and Sharing in MULTICS.» *Communications of the AC*M, Mayo 1968.

DALT96 Dalton, W., et al. *Windows NT Server 4: Security, Troubleshooting, and Optimization.* Indianapolis, IN: New Riders Publishing, 1996.

DASG92 Dasgupta, P.; et. al. «The Clouds Distributed Operating System.» *IEEE Compute*r, Noviembre 1992.

DATT90 Datta, A., y Ghosh, S. «Deadlock Detection in Distributed Systems.» *Proceedings, Phoenix Conference on Computers and Communication*s, Marzo 1990.

DATT92 Datta, A.; Javagal, R.; y Ghosh, S. «An Algorithm for Resource Deadlock Detection in Distributed Systems,» *Computer Systems Science and Engineerin*g, Octubre 1992.

DELL00 Dekker, E., y Newcomer, J. *Developing Windows NT Device Drivers: A Programmer's Handbook.* Reading, MA:Addison-Wesley, 2000.

DENN68 Denning, P. «The Working Set Model for Program Behavior.» *Communications of the AC*M, Mayo 1968.

DENN70 Denning, P. «Virtual Memory.» *Computing Survey*s, Septiembre 1970.

DENN80a Denning, P.; Buzen, J.; Dennis, J.; Gaines, R.; Hansen, P.; Lynch, W.; y Organick, E. «Operating Systems.» en [ARDE80].

DENN80b Denning, P. «Working Sets Past and Present.» *IEEE Transactions on Software Engineerin*g, Enero 1980.

DENN84 Denning, P., y Brown, R.«Operating Systems.» *Scientific America*n, Septiembre 1984.

DENN87 Denning, D. «An Intrusion-Detection Model.» *IEEE Transactions on Software Engineerin*g, Febrero 1987.

DIJK65 Dijkstra, E. *Cooperating Sequential Processe*s. Technological University, Eindhoven, The Netherlands, 1965. (Reimpreso en *Great Papers in Computer Scienc*e, P. Laplante, ed., IEEE Press, New York, NY, 1996.) También reimpreso en [BRIN01].

DIJK71 Dijkstra, E. «Hierarchical Ordering of sequential Processes.» *Acta informatic*a, Volume 1, Number 2, 1971. Reimpreso en [BRIN01].

DIMI98 Dimitoglou, G. «Deadlocks and Methods for Their Detection, Prevention, and Recovery in Modern Operating Systems.» *Operating Systems Revie*w, Julio 1998.

DONA01 Donahoo, M., y Clavert, K. *The Pocket Guide to TCP/IP Sockets.* San Francisco, CA: Morgan Kaufmann, 2001.

DOUG89 Douglas, F., y Ousterhout, J. «Process Migration in Sprite: A Status Report.» *Newsletter of the IEEE Computer Society Technical Committee on Operating System*s, Invierno 1989.

DOUG91 Douglas, F., y Ousterhout, J. «Transparent Process Migration: Design Alternatives and the Sprite Implementation.» *Software Practice and Experienc*e, Agosto 1991.

DOWD93 Dowdy, L., y Lowery, C. *P.S. to Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1993.

DUBE98 Dube, R. *A Comparison of the Memory Management Sub-Systems in FreeBSD and Linux.* Technical Report CS-TR-3929, University of Maryland, 25 Septiembre, 1998.

EAGE86 Eager, D.; Lazowska, E.; y Zahnorjan, J. «Adaptive Load Sharing in Homogeneous Distributed Systems.» *IEEE Transactions on Software Engineerin*g, Mayo 1986.

ECKE95 Eckerson, W. «Client Server Architecture.» *Network World Collaboratio*n, Invierno 1995.

EFF98 Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wire-tap Politics, and Chip Design.* Sebastopol, CA: O'Reilly, 1998

ENGE80 Enger, N., y Howerton, P. *Computer Security.* New York: Amacom, 1980.

ESKI90 Eskicioglu, M. «Design Issues of Process Migration Facilities in Distributed Systems.» *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environment*s, Verano 1990.

FEIT90a Feitelson, D., y Rudolph, L. «Distributed Hierarchical Control for Parallel Processing.» *Computer*, Mayo 1990.

FEIT90b Feitelson, D., y Rudolph, L. «Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control.» *Proceedings, 1990 International Conference on Parallel Processing*, Agosto 1990.

FERR83 Ferrari, D., y Yih, Y. «VSWS: The Variable-Interval Sampled Working Set Policy.» *IEEE Transactions on Software Engineering*, Mayo 1983.

FIDG96 Fidge, C. «Fundamentals of Distributed System Observation.» *IEEE Software*, Noviembre 1996.

FINK88 Finkel, R. *An Operating Systems Vade Mecum.* Englewood Cliffs, NJ: Prentice Hall, 1988.

FINK89 Finkel, R. «The Process Migration Mechanism of Charlotte.» *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems* , Invierno 1989.

FINK97 Finkel, R. «What is an Operating System.» In [TUCK97].

FLYN72 Flynn, M. «Computer Organizations and Their Effectiveness.» *IEEE Transactions on Computers*, Septiembre 1972.

FOLK98 Folk, M., y Zoellick, B. *File Structures: An Object-Oriented Approach with* C++. Reading, MA:Addison-Wesley, 1998.

FORR97 Forrest, S.; Hofmeyr, S.; y Somayoaji, A. «Computer Immunology.» *Communications of the ACM*, Octubre 1997.

FRAN97 Franz, M. «Dynamic Linking of Software Components.» *Computer*, Marzo 1997.

FRIE96 Friedman, M. «RAID Keeps Going and Going and…» *IEEE Spectrum*, Abril 1996.

GALL00 Galli, D. *Distributed Operating Systems: Concepts and Practice*. Upper Saddle River, NJ:Prentice Hall, 2000.

GANA98 Ganapathy, N., y Schimmel, C. «General Purpose Operating System Support for MultiplePage Sizes.» *Proceedings, USENIX Symposium*, 1998.

GARG02 Garg, V. *Elements of Distributed Computing.* New York: Wiley, 2002.

GAUD00 Gaudin, S. «The Omega Files.» *Network World*, Junio 26, 2000.

GEHR87 Gehringer, E.; Siewiorek, D.; y Segall, Z. *Parallel Processing: The Experience.*Bedford, MA: Digital Press, 1987.

GIBB87 Gibbons, P. «A Stub Generator for Multilanguage RPC in Heterogeneous Environments.» *IEEE Transactions on Software Engineering*, Enero 1987.

GING90 Gingras, A. «Dining Philosophers Revisited.» *ACM SIGCSE Bulletin*, Septiembre 1990.

GOLD89 Goldman, P. «Mac VM Revealed.» *Byte*, Noviembre 1989.

GOLL99 Gollmann, D. *Computer Security.* New York: Wiley, 1999.

GOOD94 Goodheart, B., y Cox, J. *The Magic Garden Explained: The Internals of UNIX SystemV Release 4*. Englewood Cliffs, NJ: Prentice Hall, 1994.

GOPA85 Gopal, I. «Prevention of Store-and-Forward Deadlock in Computer Networks.» *IEEE Transactions on Communications*, Diciembre 1985.

GOYE99 Goyeneche, J., y Souse, E. «Loadable Kernel Modules.» *IEEE Software*, Enero/Febrero 1999.

GRAY97 Gray, J. *Interprocess Communications in Unix: The Nooks and Crannies.* Upper Saddle River, NJ: Prentice Hall, 1997.

GROS86 Grosshans, D. *File Systems: Design and Implementation.* Englewood Cliffs, NJ: Prentice Hall, 1986.

GUPT78 Gupta, R., y Franklin, M. «Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison.» *IEEE Transactions on Computer*s, Agosto 1978.

GUYN88 Guynes, J. «Impact of System Response Time on State Anxiety.» *Communications of the AC*M, Marzo 1988.

HALD91 Haldar, S., y Subramanian, D. «Fairness in Processor Scheduling in Time Sharing Systems» *Operating Systems Revie*w, Enero 1991.

HALL01 Hall, B. *Beej's Guide to Network Programming Using Internet Sockets.* 2001. http://www.ecst.csuchico.edu/~beej/guide/net/html/

HARL01 Harley, D.; Slade, R.; y Gattiker, U. *Viruses Revealed.* New York: Osborne/McGraw-Hill, 2001.

HART97 Hartig, H., et al. «The Performance of a µ-Kernel-Based System.» *Proceedings, Sixteenth ACM Symposium on Operating Systems Principle*s, Diciembre 1997.

HATF72 Hatfield, D. «Experiments on Page Size, Program Access Patterns, and Virtual Memory Performance.» *IBM Journal of Research and Developmen*t, Enero 1972.

HENN02 Hennessy, J., y Patterson, D. *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 2002.

HENR84 Henry, G. «The Fair Share Scheduler.» *AT&T Bell Laboratories Technical Journa*l, Octubre 1984.

HERL90 Herlihy, M. «A Methodology for Implementing Highly Concurrent Data Structures,» *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programmin*g, Marzo 1990.

HOAR74 Hoare, C. «Monitors: An Operating System Structuring Concept.» *Communications of the AC*M, Octubre 1974.

HOAR85 Hoare, C. *Communicating Sequential Processes.* Englewood Cliffs, NJ: Prentice Hall, 1985.

HOFR90 Hofri, M. «Proof of a Mutual Exclusion Algorithm.» *Operating Systems Revie*w, Enero 1990.

HOLT72 Holt, R. «Some Deadlock Properties of Computer Systems.» *Computing Survey*s, Septiembre 1972.

HONG89 Hong, J.; Tan, X.; y Towsley, D. «A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System.» *IEEE Transactions on Computer*s, Diciembre 1989.

HOWA73 Howard, J. «Mixed Solutions for the Deadlock Problem.» *Communications of the AC*M, Julio 1973.

HP96 Hewlett Packard. *White Paper on Clustering.* Junio 1996.

HUCK83 Huck, T. *Comparative Analysis of Computer Architectures.* Stanford University Technical Report Number 83-243, Mayo 1983.

HUCK93 Huck, J., y Hays, J. «Architectural Support for Translation Table Management in Large Address Space Machines.» *Proceedings of the 20th Annual International Symposium on Computer Architectur*e, Mayo 1993.

HWAN99 Hwang, K, et al. «Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space.» *IEEE Concurrency*, Enero-Marzo 1999.

HYMA66 Hyman, H. «Comments on a Problem in Concurrent Programming Control.» Communications of the ACM, Enero 1966.

IBM86 IBM National Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook.* Dallas Systems Center Technical Bulletin G320-0597, Junio 1986.

INSO02a Insolvibile, G. «Inside the Linux Packet Filter.» *Linux Journal*, Febrero, 2002.

INSO02b Insolvibile, G. «Inside the Linux Packet Filter, Part II.» *Linux Journal*, Marzo, 2002.

ISLO80 Isloor, S., y Marsland, T. «The Deadlock Problem: An Overview.» *Computer*, Septiembre 1980.

IYER01 Iyer, S., y Druschel, P. «Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O.» *Proceedings, 18th ACM Symposium on Operating Systems Principles*, Octubre 2001.

JACO98a Jacob, B., y Mudge, T. «Virtual Memory: Issues of Implementation.» *Computer*,Junio 1998.

JACO98b Jacob, B., y Mudge, T. «Virtual Memory in Contemporary Microprocessors.» *IEEE Micro*, Agosto 1998.

JOHN91 Johnston, B.; Javagal, R.; Datta, A.; y Ghosh, S. «A Distributed Algorithm for Resource Deadlock Detection.» *Proceedings, Tenth Annual Phoenix Conference on Computers and Communications*, Marzo 1991.

JOHN92 Johnson, T., y Davis, T. «Space Efficient Parallel Buddy Memory Management.» *Proceedings, Third International Conference on Computers and Information*, Mayo 1992.

JONE80 Jones, S., y Schwarz, P. «Experience Using Multiprocessor Systems—A Status Report.» *Computing Surveys*, Junio 1980.

JONE97 Jones, M. «What Really Happened on Mars?» http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html, 1997.

JUL88 Jul, E.; Levy, H.; Hutchinson, N.; y Black, A. «Fine-Grained Mobility in the Emerald System.» *ACM Transactions on Computer Systems*, Febrero 1988.

JUL89 Jul, E. «Migration of Light-Weight Processes in Emerald.» *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Invierno 1989.

KANG98 Kang, S., y Lee, J. «Analysis and Solution of Non-Preemptive Policies for Scheduling Readers and Writers.» *Operating Systems Review*, Julio 1998.

KAPP00 Kapp, C. «Managing Cluster Computers.» *Dr. Dobb's Journal*, Julio 2000.

KATZ89 Katz, R.; Gibson, G.; y Patterson, D. «Disk System Architecture for High Performance Computing.» *Proceedings of the IEEE*, Diciembre 1989.

KAY88 Kay, J., y Lauder, P. «A Fair Share Scheduler.» *Communications of the ACM*, Enero 1988.

KEFE92 Kefir, M., y Smolka, S. «OSP: An Environment for Operating System Projects.» *Operating Systems Review*, Abril 1992.

KENT00 Kent, S. «On the Trail of Intrusions into Information Systems.» *IEEE Spectrum*, Diciembre 2000.

KEPH97a Kephart, J.; Sorkin, G.; Chess, D.; y White, S. «Fighting Computer Viruses.» *Scientific American*, Noviembre 1997.

KEPH97b Kephart, J.; Sorkin, G.; Swimmer, B.; y White, S. «Blueprint for a Computer Immune System.» *Proceedings, Virus Bulletin International Conferenc*e, Octubre 1997.

KESS92 Kessler, R., y Hill, M. «Page Placement Algorithms for Large Real-Indexed Caches.» *ACM Transactions on Computer System*s, Noviembre 1992.

KHAL93 Khalidi, Y.; Talluri, M.; Williams, D.; y Nelson, M. «Virtual Memory Support for Multiple Page Sizes.» *Proceedings, Fourth Workshop on Workstation Operating System*s, Octubre 1993.

KIFE92 Kifer, M., y Smolka, S. «OSP: An Environment for Operating Systems Projects.» *ACM Operating Systems Revie*w, Octubre 1992.

KILB62 Kilburn, T.; Edwards, D.; Lanigan, M.; y Sumner, F. «One-Level Storage System.» *IRE Transaction*s, Abril 1962.

KLEI90 Klein, D. «Foiling the Cracker: A Survey of, and Improvements to, Password Security.» *Proceedings, UNIX Security Workshop I*I, Agosto 1990.

KLEI95 Kleiman, S. «Interrupts as Threads.» *Operating System Revie*w, Abril 1995.

KLEI96 Kleiman, S.; Shah, D.; y Smallders, B. *Programming with Threads.* Upper Saddle River, NJ: Prentice Hall, 1996.

KLEI04 Kleinrock, L. *Queuing Systems, Volume Three: Computer Applications.* New York: Wiley, 2004.

KNUT71 Knuth, D. «An Experimental Study of FORTRAN Programs.» *Software Practice and Experienc*e, Vol. 1, 1971.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Reading, MA:Addison-Wesley, 1997.

KORS90 Korson, T., y McGregor, J. «Understanding Object-Oriented: A Unifying Paradigm.» *Communications of the AC*M, Septiembre 1990.

KRIS94 Krishna, C., y Lee, Y., eds. «Special Issue on Real-Time Systems.» *Proceedings of the IEEE*, Enero 1994.

KRIS03 Krishnaprasad, S. «Concurrent/Distributed Programming Illustrated Using the Dining Philosophers Problem.» *The Journal of Computing in Small College*s, Abril 2003.

KRON90 Kron, P. «A Software Developer Looks at OS/2.» *Byte*, Agosto 1990.

LAMP74 Lamport, L. «A New Solution to Dijkstra's Concurrent Programming Problem.» *Communications of the AC*M, Agosto 1974.

LAMP78 Lamport, L. «Time, Clocks, and the Ordering of Events in a Distributed System.» *Communications of the AC*M, Julio 1978.

LAMP80 Lampson, B., y Redell D. «Experience with Processes and Monitors in Mesa.» *Communications of the AC*M, Febrero 1980.

LAMP86 Lamport, L. «The Mutual Exclusion Problem.» *Journal of the AC*M, Abril 1986.

LAMP91 Lamport, L. «The Mutual Exclusion Problem Has Been Solved.» *Communications of the AC*M, Enero 1991.

LARO92 LaRowe, R.; Holliday, M.; y Ellis, C. «An Analysis of Dynamic Page Placement an a NUMA Multiprocessor.» Proceedings, *1992 ACM SIGMETRICS and Performance '9*2, Junio 1992.

LEBL87 LeBlanc, T., y Mellor-Crummey, J. «Debugging Parallel Programs with Instant Replay.» *IEEE Transactions on Computer*s, Abril 1987.

LEE93 Lee, Y., y Krishna, C., eds. *Readings in Real-Time Systems.* Los Alamitos, CA: IEEE Computer Society Press, 1993.

LELA86 Leland,W., y Ott, T. «Load-Balancing Heuristics and Process Behavior.» *Proceedings, ACM SigMetrics Performance 1986 Conferenc*e, 1986.

LERO76 Leroudier, J., y Potier, D. «Principles of Optimality for Multiprogramming.» *Proceedings, International Symposium on Computer Performance Modeling, Measurement, and Evaluatio*n, Marzo 1976.

LETW88 Letwin, G. *Inside OS/2.* Redmond, WA: Microsoft Press, 1988.

LEUT90 Leutenegger, S., y Vernon, M. «The Performance of Multiprogrammed Multiprocessor Scheduling Policies.» *Proceedings, Conference on Measurement and Modeling of Computer System*s, Mayo 1990.

LEVI99 Levine, J. *Linkers and Loaders.* New York: Elsevier Science and Technology, 1999.

LEVI03a Levine, G. «Defining Deadlock.» *Operating Systems Revie*w, Enero 2003.

LEVI03b Levine, G. «Defining Deadlock with Fungible Resources.» *Operating Systems Revie*w, Julio 2003.

LEWI96 Lewis, B., y Berg, D. *Threads Primer.* Upper Saddle River, NJ: Prentice Hall, 1996.

LIED95 Liedtke, J. «On -Kernel Construction.» *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principle*s, Diciembre 1995.

LIED96a Liedtke, J. «Toward Real Microkernels.» *Communications of the AC*M, Septiembre 1996.

LIED96b Liedtke, J. «Microkernels Must and Can Be Small.» *Proceedings, Fifth International Workshop on Object Orientation in Operating System*s, Octubre 1996.

LIND04 Lindsley, R. «What's New in the 2.6 Scheduler.» *Linux Journa*l, Marzo 2004.

LIST93 Lister, A., y Eager, R. *Fundamentals of Operating Systems.* New York: Springer-Verlag, 1993.

LIU73 Liu, C., y Layland, J. «Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment.» Journal of the ACM, Febrero 1973.

LIU00 Liu, J. *Real-Time Systems.* Upper Saddle River, NJ: Prentice Hall, 2000.

LIVA90 Livadas, P. *File Structures: Theory and Practice.* Englewood Cliffs, NJ: Prentice Hall, 1990.

LOVE04a Love, R. *Linux Kernel Development.* Indianapolis, IN: Sams Publishing, 2004.

LOVE04b Love, R. «I/O Schedulers.» *Linux Journa*l, Febrero 2004.

LYNC96 Lynch, N. *Distributed Algorithms.* San Francisco, CA: Morgan Kaufmann, 1996.

MADS93 Madsen, J. «World Record in Password Checking.» *USENET, comp.security.misc news-grou*p, Agosto 18, 1993.

MAEK87 Maekawa, M.; Oldehoeft,A.; y Oldehoeft, R. *Operating Systems: Advanced Concepts.* Menlo Park, CA: Benjamin Cummings, 1987.

MAJU88 Majumdar, S.; Eager, D.; y Bunt, R. «Scheduling in Multiprogrammed Parallel Systems.» *Proceedings, Conference on Measurement and Modeling of Computer System*s, Mayo 1988.

MART88 Martin, J. *Principles of Data Communication.* Englewood Cliffs, NJ: Prentice Hall, 1988.

MASS97 Massiglia, P. (editor). *The RAID Book: A Storage System Technology Handbook.* St. Peter, MN: The Raid Advisory Board, 1997.

MAUR01 Mauro, J., y McDougall, R. *Solaris Internals: Core Kernel Architecture.* Palo Alto, CA: Sun Microsystems Press, 2001.

MCHU00 McHugh, J.; Christie, A.; y Allen, J. «The Role of Intrusion Detection Systems.» *IEEE Software*, Septiembre/Octubre 2000.

MCKU96 McKusick, M.; Bostic, K.; Karels, M.; y Quartermain, J. *The Design and Implementation of the 4.4BSD UNIX Operating System.* Reading, MA:Addison-Wesley, 1996.

MEE96a Mee, C., y Daniel, E. eds. *Magnetic Recording Technology.* New York: McGraw-Hill, 1996.

MEE96b Mee, C., y Daniel, E. eds. *Magnetic Storage Handbook.* New York: McGraw-Hill, 1996.

MESS96 Messer, A., y Wilkinson, T. «Components for Operating System Design.» *Proceedings, Fifth International Workshop on Object Orientation in Operating Systems*, Octubre 1996.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design.* New York: McGraw-Hill, 1992.

MILO00 Milojicic, D.; Douglis, F.; Paindaveine, Y.; Wheeler, R.; y Zhou, S. «Process Migration.» *ACM Computing Surveys*, Septiembre 2000.

MORG92 Morgan, K. «The RTOS Difference.» *Byte*, Agosto 1992.

MOSB02 Mosberger, D., y Eranian, S. *IA-64 Linux Kernel*. Upper Saddle River, NJ: Prentice Hall, 2002.

MS96 Microsoft Corp. *Microsoft Windows NT Workstation Resource Kit.* Redmond, WA: Microsoft Press, 1996.

NACH97 Nachenberg, C. «Computer Virus-Antivirus Coevolution.» *Communications of the ACM*, Enero 1997.

NEHM75 Nehmer, J. «Dispatcher Primitives for the Construction of Operating System Kernels.» *Acta Informatica*, vol 5, 1975.

NELS88 Nelson, M.; Welch, B.; y Ousterhout, J. «Caching in the Sprite Network File System.» *ACM Transactions on Computer Systems*, Febrero 1988.

NELS91 Nelson, G. *Systems Programming with Modula-3.* Englewood Cliffs, NJ: Prentice Hall, 1991.

NG98 Ng, S. «Advances in Disk Technology: Performance Issues.» *Computer*, Mayo 1989.

NUTT94 Nuttal, M. «A Brief Survey of Systems Providing Process or Object Migration Facilities.»*Operating Systems Review*, Octubre 1994.

NUTT04 Nutt, G. *Operating System.* Reading, MA:Addison-Wesley, 2004.

OUST85 Ousterhout, J., et al. «A Trace-Drive Analysis of the UNIX 4.2 BSD File System.» *Proceedings, Tenth ACM Symposium on Operating System Principles*, 1985.

OUST88 Ousterhout, J., et al. «The Sprite Network Operating System.» *Computer,* Febrero 1988.

PAI00 Pai, V.; Druschel, P.; y Zwaenepoel,W. «IO-Lite: A Unified I/O Buffering and Caching System.» *ACM Transactions on Computer Systems*, Febrero 2000.

PANW88 Panwar, S.; Towsley, D.; y Wolf, J. «Optimal Scheduling Policies for a Class of Queues with Customer Deadlines in the Beginning of Service.» *Journal of the ACM*, Octubre 1988.

PATT82 Patterson, D., y Sequin, C. «A VLSI RISC.» *Computer*, Septiembre 1982.

PATT85 Patterson, D. «Reduced Instruction Set Computers.» *Communications of the ACM*, Enero 1985.

PATT88 Patterson, D.; Gibson, G.; y Katz, R.«A Case for Redundant Arrays of Inexpensive Disks (RAID).» *Proceedings, ACM SIGMOD Conference of Management of Data*, Junio 1988.

PATT98 Patterson, D., y Hennessy, J. *Computer Organization and Design:The Hardware/Software Interface.* San Mateo, CA: Morgan Kaufmann, 1998.

PEIP03 Pieprzyk, J.; Hardjono, T.; y Seberry, J. *Fundamentals of Computer Security.* New York: Springer, 2003.

PERR03 Perrine, T. «The End of crypt() Passwords Please?» *;logi*n, Diciembre 2003.

PETE77 Peterson, J., y Norman, T. «Buddy Systems.» *Communications of the AC*M, Junio 1977.

PETE81 Peterson, G. «Myths About the Mutual Exclusion Problem.» *Information Processing Letter*s, Junio 1981.

PFLE97 Pfleeger, C. *Security in Computing.* Upper Saddle River, NJ: Prentice Hall PTR, 1997.

PINK89 Pinkert, J., y Wear, L. *Operating Systems: Concepts, Policies, and Mechanisms.* Englewood Cliffs, NJ: Prentice Hall, 1989.

PIZZ89 Pizzarello, A. «Memory Management for a Large Operating System.» *Proceedings, International Conference on Measurement and Modeling of Computer System*s, Mayo 1989.

POPE85 Popek, G., y Walker, B. *The LOCUS Distributed System Architectur*e, Cambridge, MA: MIT Press, 1985.

PORR92 Porras, P. *STAT: A State Transition Analysis Tool for Intrusion Detection.* Master's Thesis, University of California at Santa Barbara, Julio 1992.

PRAM84 Pramanik, S., y Weinberg, B. «The Implementation Kit with Monitors,» *SIGPLAN Notice*s, Number 9, 1984.

PRZY88 Przybylski, S.; Horowitz, M.; y Hennessy, J. «Performance Trade-offs in Cache Design.» *Proceedings, Fifteenth Annual International Symposium on Computer Architectur*e, Junio 1988.

RAMA94 Ramamritham, K., y Stankovic, J. «Scheduling Algorithms and Operating Systems Support for Real-Time Systems.» *Proceedings of the IEE*E, Enero 1994.

RASH88 Rashid, R., et al. «Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.» *IEEE Transactions on Computer*s, Agosto 1988.

RAYN86 Raynal, M. *Algorithms for Mutual Exclusion.* Cambridge, MA: MIT Press, 1986.

RAYN88 Raynal, M. Distributed Algorithms and Protocols. New York: Wiley, 1988.

RAYN90 Raynal, M., y Helary, J. Synchronization and Control of Distributed Systems and Programs. New York: Wiley, 1990.

RICA81 Ricart, G., y Agrawala,A.«An Optimal Algorithm for Mutual Exclusion in Computer Networks.» Communications of the ACM, Enero 1981 (Corrigendum en Communications of the ACM, Septiembre 1981).

RICA83 Ricart, G., y Agrawala,A.«Author's Response to 'On Mutual Exclusion in Computer Networks' by Carvalho y Roucairol.» Communications of the ACM, Febrero 1983.

RIDG97 Ridge, D., et al. «Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs.» Proceedings, IEEE Aerospace, 1997.

RITC74 Ritchie, D., y Thompson, K. «The UNIX Time-Sharing System.» Communications of the ACM, Julio 1974.

RITC78 Ritchie, D. «UNIX Time-Sharing System: A Retrospective.» The Bell System Technical Journal, Julio-Agosto 1978.

ROBI90 Robinson, J., y Devarakonda, M. «Data Cache Management Using Frequency-Based Replacement.» Proceedings, Conference on Measurement and Modeling of Computer Systems, Mayo 1990.

RODR02 Rodriguez, A., et al. TCP/IP Tutorial and Technical Overview. Upper Saddle River: NJ: Prentice Hall, 2002.

ROSC03 Rosch, W. The Winn L. Rosch Hardware Bible. Indianapolis, IN: Sams, 2003.

ROSE78 Rosenkrantz, D.; Stearns, R.; y Lewis, P. «System Level Concurrency Control in Distributed Database Systems.» ACM Transactions on Database Systems, Junio 1978.

RUBI97 Rubini, A. «The Virtual File System in Linux.» Linux Journal, Mayo 1997.

RUDO90 Rudolph, B. «Self-Assessment Procedure XXI: Concurrency.» Communications of the ACM, Mayo 1990.

SAND94 Sandhu, R, y Samarati, P. «Access Control: Principles and Practice.» IEEE Communications, Septiembre 1994.

SATY81 Satyanarayanan, M. y Bhandarkar, D. «Design Trade-Offs in VAX-11 Translation Buffer Organization.» Computer, Diciembre 1981.

SAUE81 Sauer, C, y Chandy, K. Computer Systems Performance Modeling. Englewood Cliffs, NJ: Prentice Hall, 1981.

SCHA62 Schay, G., y Spruth, W. «Analysis of a File Addressing Method.» Communications of the ACM, Agosto 1962.

SCHM97 Schmidt, D. «Distributed Object Computing.» IEEE Communications Magazine, Febrero 1997.

SCHN96 Schneier, B. Applied Cryptography. New York: Wiley, 1996.

SCHN99 Schneier, B. «The Trojan Horse Race.» Communications of the ACM, Septiembre 1999.

SCHW96 Schwaderer, W., y Wilson, A. Understanding I/O Subsystems. Milpitas, CA: Adaptec Press, 1996.

SELT90 Seltzer, M.; Chen, P.; y Ousterhout, J. «Disk Scheduling Revisited.» Proceedings, USENIX Winter Technical Conference, Enero 1990.

SEVC96 Sevcik, P. «Designing a High-Performance Web Site.» Business Communications Review, Marzo 1996.

SHA90 Sha, L.; Rajkumar, R.; y Lehoczky, J. «Priority Inheritance Protocols: An Approach to Real-Time Synchronization.» IEEE Transactions on Computers, Septiembre 1990.

SHA91 Sha, L.; Klein, M.; y Goodenough, J. «Rate Monotonic Analysis for Real-Time Systems.» en [TILB91].

SHA94 Sha, L.; Rajkumar, R.; y Sathaye, S. «Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems.» Proceedings of the IEEE, Enero 1994.

SHEN02 Shene, C. «Multithreaded Programming Can Strengthen an Operating Systems Course.» *Computer Science Education Journa*l, Diciembre 2002.

SHIV92 Shivaratri, N.; Krueger, P.; y Singhal, M.«Load Distributing for Locally Distributed Systems.» *Compute*r, Diciembre 1992.

SHNE84 Shneiderman, B. «Response Time and Display Rate in Human Performance with Computers.» *ACM Computing Survey*s, Septiembre 1984.

SHOR75 Shore, J. «On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies.» *Communications of the AC*M, Agosto, 1975.

SHUB03 Shub, C. «A Unified Treatment of Deadlock.» *Journal of Computing in Small College*s, Octubre 2003. Disponible en la biblioteca digital de ACM.

SILB98 Silberschatz, A., y Galvin, P. *Operating System Concepts*. Reading, MA: Addison-Wesley, 1998.

SILB04 Silberschatz, A.; Galvin, P.; y Gagne, G. *Operating System Concepts with Java*. Reading, MA:Addison-Wesley, 2004.

SING94a Singhal, M., y Shivaratri, N. *Advanced Concepts in Operating Systems*. New York: McGraw-Hill, 1994.

SING94b Singhal, M. «Deadlock Detection in Distributed Systems.» En [CASA94].

SINH97 Sinha, P. *Distributed Operating Systems*. Piscataway, NJ: IEEE Press, 1997.

SMIT82 Smith, A. «Cache Memories.» *ACM Computing Survey*s, Septiembre 1982.

SMIT83 Smith, D. «Faster Is Better: A Business Case for Subsecond Response Time.» *Computerworl*d, Abril 18, 1983.

SMIT85 Smith, A.«Disk Cache—Miss Ratio Analysis and Design Considerations.» *ACM Transactions on Computer System*s, Agosto 1985.

SMIT88 Smith, J. «A Survey of Process Migration Mechanisms.» *Operating Systems Revie*w, Julio 1988.

SMIT89 Smith, J. «Implementing Remote *fork()* with Checkpoint/restart.» *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Winter 1989.

SOLO00 Solomon, D. *Inside Microsoft Windows 2000*. Redmond, WA: Microsoft Press, 2000.

SPAF92 Spafford, E. «Observing Reusable Password Choices.» *Proceedings, UNIX Security Symposium II*I, Septiembre 1992.

STAL03a Stallings, W. *Computer Organization and Architecture, 6th ed*. Upper Saddle River, NJ: Prentice Hall, 2003.

STAL03b Stallings, W. *Cryptography and Network Security: Principles and Practice, Third Edition*. Upper Saddle River, NJ: Prentice Hall, 2003.

STAL04 Stallings, W. *Computer Networking with Internet Protocols and Technology*. Upper Saddle River: NJ: Prentice Hall, 2004.

STAN89 Stankovic, J., y Ramamrithan, K. «The Spring Kernel: A New Paradigm for Real-Time Operating Systems.» *Operating Systems Revie*w, Julio 1989.

STAN93 Stankovic, J., y Ramamritham, K., eds. *Advances in Real-Time Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1993.

STEP93 Stephenson, P. «Preventive Medicine.» *LAN Magazin*e, Noviembre 1993.

STON93 Stone, H. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1993.

STRE83 Strecker, W. «Transient Behavior of Cache Memories.» *ACM Transactions on Computer Systems,* Noviembre 1983.

STRO88 Stroustrup, B. «What is Object-Oriented Programming?» *IEEE Softwar*e, Mayo 1988.

SNYD93 Snyder, A. «The Essence of Objects: Concepts and Terms.» *IEEE Softwar*e, Enero 1993.

SUZU82 Suzuki, I., y Kasami, T. «An Optimality Theory for Mutual Exclusion Algorithms in Computer Networks.» *Proceedings of the Third International Conference on Distributed Computing Systems*, Octubre 1982.

TALL92 Talluri, M.; Kong, S.; Hill, M.; y Patterson, D. «Tradeoffs in Supporting Two Page Sizes.» *Proceedings of the 19th Annual International Symposium on Computer Architectur*e, Mayo 1992.

TAMI83 Tamir, Y., y Sequin, C. «Strategies for Managing the Register File in RISC.» *IEEE Transactions on Computer*s, Noviembre 1983.

TANE78 Tanenbaum, A. «Implications of Structured Programming for Machine Architecture.» *Communications of the AC*M, Marzo 1978.

TANE97 Tanenbaum, A., y Woodhull, A. *Operating Systems: Design and Implementation.* Upper Saddle River, NJ: Prentice Hall, 1997.

TANE01 Tanenbaum, A. *Modern Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 2001.

TAIV96 Taivalsaari, A. «On the Nature of Inheritance.» *ACM Computing Survey*s, Septiembre 1996.

TEL01 Tel, G. *Introduction to Distributed Algorithms.* Cambridge: Cambridge University Press, 2001.

THAD81 Thadhani, A. «Interactive User Productivity.» *IBM Systems Journa*l, No. 1, 1981.

THOM84 Thompson, K. «Reflections on Trusting Trust (Deliberate Software Bugs).» *Communications of the AC*M, Agosto 1984.

TILB91 Tilborg, A., y Koob, G.. eds. *Foundations of Real-Time Computing: Scheduling and Resource Management.* Boston: Kluwer Academic Publishers, 1991.

TIME90 Time, Inc. *Computer Security, Understanding Computers Serie*s. Alexandria, VA: Time-Life Books, 1990.

TIME02 TimeSys Corp. «Priority Inversion: Why You Care and What to Do About It» *TimeSys White Pape*r, 2002. http://www.techonline.com/community/ ed_resource/tech_paper/21779

TUCK89 Tucker, A., y Gupta, A. «Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors.» *Proceedings, Twelfth ACM Symposium on Operating Systems Principle*s, Diciembre 1989.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers.* Upper Saddle River, NJ: Prentice Hall, 1996.

VINO97 Vinoski, S. «CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.» *IEEE Communications Magazin*e, Febrero 1997.

WALK89 Walker, B., y Mathews, R. «Process Migration in AIX's Transparent Computing Facility.» *Newsletter of the IEEE Computer Society Technical Committee on Operating System*s, Invierno 1989.

WARD80 Ward, S. «TRIX: A Network-Oriented Operating System.» *Proceedings, COMP/CON '8*0, 1980.

WARR91 Warren, C. «Rate Monotonic Scheduling.» *IEEE Micr*o, Junio 1991.

WAYN94a Wayner, P. «Small Kernels Hit it Big.» *Byt*e, Enero 1994.

WAYN94b Wayner, P. «Objects on the Marzo.» *Byt*e, Enero 1994.

WEIZ81 Weizer, N. «A History of Operating Systems.» *Datamatio*n, Enero 1981.

WEND89 Wendorf, J.; Wendorf, R.; y Tokuda, H. «Scheduling Operating System Processing on Small-Scale Microprocessors.» *Proceedings, 22nd Annual Hawaii International Conference on System Scienc*e, Enero 1989.

WHIT99 White, S.; Swimmer, M.; Pring, E.; Arnold, B.; Chess, D.; y Morar, J. *Anatomy of a Commercial-Grade Immune System.* IBM White Paper, 1999. www.research.ibm.com/antivirus/SciPapers.htm.

WIED87 Wiederhold, G. *File Organization for Database Design.* New York: McGraw-Hill, 1987.

WOOD86 Woodside, C. «Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers.» *IEEE Transactions on Software Engineerin*g, Octubre 1986

WOOD89 Woodbury, P. et al. «Shared Memory Multiprocessors: The Right Approach to Parallel Processing.» *Proceedings, COMPCON Spring '8*9, Marzo 1989.

WORT94 Worthington, B.; Ganger, G.; y Patt, Y. «Scheduling Algorithms for Modern Disk Drives.» *ACM SiGMETRIC*S, Mayo 1994.

WRIG95 Wright, G., y Stevens, W. *TCP/IP Illustrated, Volume 2: The Implementation.* Reading, MA:Addison-Wesley, 1995.

YOUN87 Young, M., et. al. «The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.» *Proceedings of the Eleventh ACM Symposium on Operating Systems Principle*s, Diciembre 1987.

ZAHO90 Zahorjan, J., y McCann, C. «Processor Scheduling in Shared Memory Multiprocessors.» *Proceedings, Conference on Measurement and Modeling of Computer System*s, Mayo 1990.

ZAJC93 Zajcew, R., et al. «An OSF/1 UNIX for Massively Parallel Multicomputers.» *Proceedings, Winter USENIX Conferenc*e, Enero 1993.

ZEAD97 Zeadally, S. «An Evaluation of the Real-Time Performance of SVR4.0 and SVR4.2.» *Operating Systems Revie*w, Enero 1977.

Acrónimos

AES	*Advanced Encryption Standard* (Estándar de cifrado avanzado)
API	*Application Programming Interface* (Interfaz de programación de aplicaciones)
CD	*Compact Disk* (Disco compacto)
CORBA	*Common Object Request Broker Architecture* (Arquitectura común de mediador de solicitud de objeto)
CPU	*Central Processing Unit* (Unidad central de proceso)
CTSS	*Compatible Time-Sharing System* (Sistema de tiempo compartido compatible)
DES	*Data Encryption Standard* (Estándar de cifrado de datos)
DMA	*Direct Memory Access* (Acceso directo a memoria)
DVD	*Digital Versatile Disk* (Disco digital versátil)
E/S	Entrada/Salida
FAT	*File Allocation Table* (Tabla de asignación de ficheros)
FCFS	*First Come First Served* (Primero en llegar, primero en servirse)
FIFO	*First In First Out* (Primero en entrar, primero en salir)
GUI	*Graphical User Interface* (Interfaz gráfica de usuario)
IP	*Internet Protocol* (Protocolo de Internet)
IBM	*International Business Machines Corporation* (Corporación Internacional de Máquinas de Empresa)
IPC	*InterProcess Communication* (Comunicación entre procesos)
JCL	*Job Control Language* (Lenguaje de control de trabajos)
LAN	*Local Area Network* (Red de área local)
LIFO	*Last In First Out* (Último en entrar, primero en salir)
LRU	*Least Recently Used* (Menos recientemente usado)
MVS	*Multiple Virtual Storage* (Almacenamiento virtual múltiple)
NTFS	*NT File System* (Sistema de ficheros de NT)
NUMA	*Nonuniform Memory Access* (Acceso a memoria no uniforme)
ORB	*Object Request Broker* (Mediador de solicitud de objeto)
OSI	*Open Systems Interconnection* (Interconexión de sistemas abiertos)
PC	*Program Counter* (Contador de programa)
PSW	*Processor Status Word* (Palabra de estado del programa)
PCB	*Process Control Block* (Bloque de control del proceso)

RAID *Redundant Array of Independent Disks* (Vector redundante de discos independientes)

RISC *Reduced Instruction Set Computer* (Computador con un juego de instrucciones reducido)

RPC *Remote Procedure Call* (Llamada a procedimiento remoto)

SMP *Symmetric Multiprocessing or Symmetric Multiprocessor* (Multiprocesamiento simétrico o Multiprocesador simétrico)

SPOOL *Simultaneous Peripheral Operation On Line* (Operación simultánea en línea de periféricos)

SVR4 System V Release 4

TCP *Transmission Control Protocol* (Protocolo de control de transmisión)

TLB *Translation Lookaside Buffer* (*Buffer* de traducción anticipada)

UDP *User Datagram Protocol* (Protocolo de datagramas de usuario)

Í N D I C E A N A L Í T I C O

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

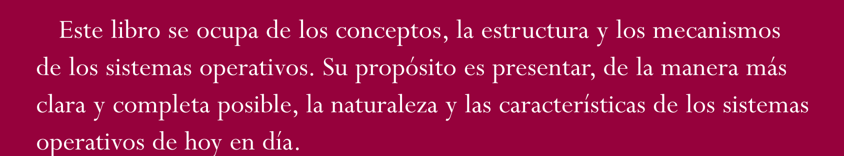
T

U

V

W

Z



Este libro se ocupa de los conceptos, la estructura y los mecanismos de los sistemas operativos. Su propósito es presentar, de la manera más clara y completa posible, la naturaleza y las características de los sistemas operativos de hoy en día.

En esta nueva edición, el autor ha intentado recoger las innovaciones y mejoras que ha habido en esta disciplina durante los cuatro años que han transcurrido desde la última edición, manteniendo un tratamiento amplio y completo de esta materia. Asimismo, varios profesores que imparten esta disciplina, así como profesionales que trabajan en este campo, han revisado en profundidad la cuarta edición. Como consecuencia de este proceso, en muchas partes del libro, se ha mejorado la claridad de la redacción y de las ilustraciones que acompañan al texto. Además, se han incluido varios problemas de carácter realista.

Además de mejoras pedagógicas y en su presentación de cara al usuario, el contenido técnico del libro se ha actualizado completamente, para reflejar los cambios actuales en esta excitante disciplina. El estudio de Linux se ha extendido significativamente, basándose en su última versión: Linux 2.6. El estudio de Windows se ha actualizado para incluir Windows XP y Windows Server 2003.

Se incluye un nuevo capítulo sobre redes, presentándose el API de Sockets. Además, se ha ampliado el tratamiento del diseño orientado a objetos.

Hay un sitio web asociado a este libro que proporciona apoyo a los estudiantes y a los profesores. El sitio incluye enlaces a otros sitios relevantes, copias originales de las transparencias de las figuras y tablas del libro en formato PDF (Adobe Acrobat), transparencias en PowerPoint e información para darse de alta en la lista de correo de Internet del libro. La página web está en **WilliamStallings.com/OS/OS5e.html**



www.pearsoneducacion.com

